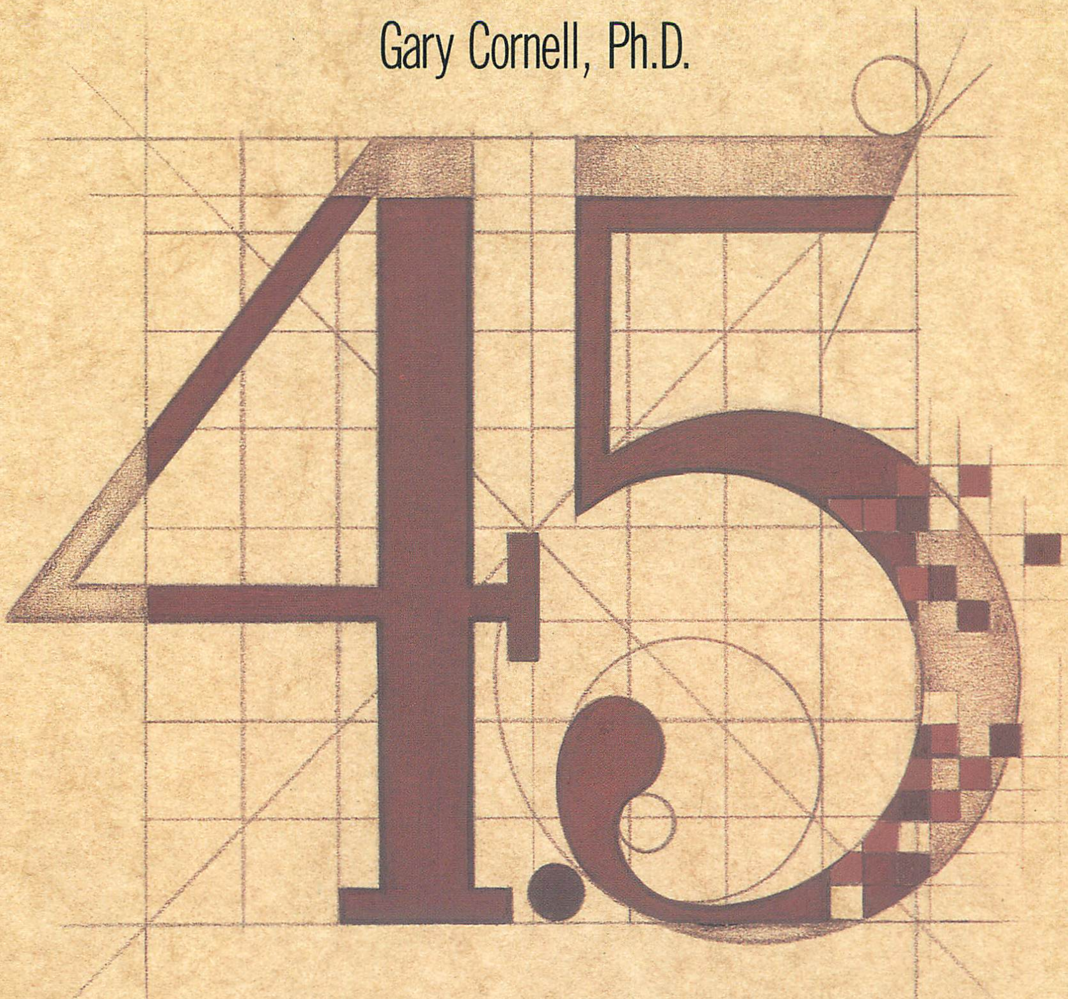
WINDCREST®/McGRAW-HILL

QuickBASIC® 4.5

Gary Cornell, Ph.D.



- Teaches structured programming in BASIC
- Covers fractals, cryptography, recursion, and more
- Includes indexing, B-trees, and ISAM
- Packed with hundreds of program examples

QuickBASIC® 4.5

QuickBASIC® 4.5

Gary Cornell, Ph.D.

Windcrest®/McGraw-Hill

FIRST EDITION
THIRD PRINTING

Library of Congress Cataloging-in-Publication Data

Cornell, Gary.
QuickBASIC 4.5 / by Gary Cornell.
p. cm.
Includes index.
ISBN 0-8306-3782-6 (paper)
1. BASIC (Computer program language) 2. Microsoft QuickBASIC (Computer program) I. Title.
QA76.73.B3C68 1991 91-9674
005.26'2—dc20 CIP

Acquisitions Editor: Stephen Moore
Technical Editor: David M. Harter
Director of Production: Katherine G. Brown
Series Design: Jaclyn J. Boone WT1

Contents

Introduction

This book is a hands-on tutorial for QuickBASIC 4.5. After a quick introduction, I move on to more sophisticated topics. It's written for people who have dabbled with either interpreted BASIC or QuickBASIC and now want to move towards mastery of QuickBASIC 4.5. Nonetheless, I hope a dedicated beginner will find it usable as well. I'll show you how QuickBASIC 4.5 is a vast extension of traditional BASICs—and how to take full advantage of these advanced features. (BASIC is still the most popular language for programming, and the advanced features in QuickBASIC will make almost any programming job easier—as well as making it more likely that it will stay the most popular language.)

I've tried hard to stress those features of QuickBASIC that make it one of the best examples of a modern structured programming language: FUNCTIONal procedures, SUBprocedures and user defined types. ("Best" because unlike many academics, I think the accessability of BASIC is a virtue not a vice.)

One way I've tried to keep your interest is to include many topics not usually covered in books at this level—or for that matter in any book on BASIC:

- An extensive treatment of recursion—including an introduction to fractals.
- Multiple methods of sorting and searching—including an introduction to binary trees.
- A treatment of elementary cryptography (keeping your files secret).

There are also over 150 programs. (Some of them are quite long. If you decide that you'd prefer not to enter them by hand, they're all available on the supplemental diskette.)

Obviously, I'd like to hear from you—especially if you have any suggestions for improvements or corrections. I have a BIX account—my name there is gcornell. Finally, you can write to me at the address given on the order form for the supplemental diskette—I don't guarantee I'll respond, but I'll certainly try.

Writing any book—especially one on microcomputers—is made easier by the tools those very same microcomputers make possible. In my case, while programming in QuickBASIC, I made extensive use of two of them. The first, INSET by Inset Systems, is responsible for the many screen dumps in this book. The second, PC TOOLS DELUXE by Central Point Software, was both a useful add-on for manipulating files and an efficient way of keeping notes. (The actual text of this book was done using WordPerfect.)

Now to the best part of writing a book—the place where the author gets to thank those who have helped him or her (for rarely is a book by one author produced alone). First off, I have to thank those people at Microsoft (whose names I unfortunately don't know) who created QuickBASIC 4.5. It's a great language. People whose names I do know at Microsoft and whose help was vital are Greg Lobdell, Lynn MacIntyre and Ray Kanamori. Hal Reinisch at IBM made it possible for me to use a PS/2. Finally, I'd like to thank all the dedicated people at the TAB division of McGraw-Hill—without their dedication (and patience!) this book would never have been possible.

1 CHAPTER

Getting started

This chapter is introductory: if you are an experienced interpreted (GW) basic programmer, it will introduce the new features of QuickBASIC (QB for short). If you've dabbled with QuickBASIC, you can probably skip it. I'll tell you what you need to run QuickBASIC 4.5 as well as how to set it up. Next, I'll briefly explain the differences between compilers and interpreters and why QuickBASIC 4.5 combines the best features of both. The last section covers some features of PC/MS-DOS that you might not be familiar with: batch files and reconfiguring (via CONFIG.SYS and AUTOEXEC.BAT) your machine.

Of course, I'm assuming that you know enough about DOS to format disks, to make copies, and to copy files. If you're new to DOS then you'll need to know these procedures before going on.

Getting started with QuickBASIC

QuickBASIC 4.5 requires an IBM PC, PC-XT, PC-AT, PS/2 or compatible. You'll need DOS 2.0 or later. You'll also need 384K and a minimum of two 360K drives (double sided/double density). One 1.2 megabyte quad density, or one 3.5 inch drive will also work. However, you really want 512K of RAM (640K is even better) and a hard disk to take full advantage of the power of QuickBASIC 4.5. On a system with only floppy disk drives you'll be forced to swap disks to take full advantage of QB. (I'll often use this abbreviation instead of QuickBASIC 4.5.)

QuickBASIC 4.5 also supports the Microsoft mouse and "mice" compatible with this. (Mice are nice, but after a while I suspect that you, like most experienced programmers, will end up preferring the various keyboard shortcuts of QB. See appendix D for a complete list of these.)

If you want to use any of the graphics features of QuickBASIC 4.5—draw circles, pie charts, or the like—then obviously you must have a graphics adapter. QB can work with almost any kind of graphic card—from the "Color Graphics adaptor" (CGA) found on the earliest versions of the IBM PC to the VGA found on the PS/2 series. QuickBASIC 4.5 can also work with the Hercules graphic card—more on how to do this later.

QuickBASIC 4.5 comes with two manuals and five 5.25" (360K) disks or three 3.5 (720K) disks. An astonishingly good Help system is available at the touch of a key when programming in QB. This help system, once you master it (and it's pretty simple to do), makes the programming much easier. There's little need for a large number of bulky manuals with QuickBASIC 4.5 because so much information is available on-line.

QuickBASIC 4.5 comes with an automated SETUP program found on the disk aptly titled SETUP (or SETUP/MICROSOFT QuickBASIC 4.5 EXPRESS for 3.5 inch disks) that you shouldn't have much trouble using. One thing to check is the late breaking news that Microsoft puts on the distribution disks. The most important (if it's there) is a short file that will be called, naturally enough, READ THIS.NOW. I found mine on the disk marked SETUP mentioned earlier.

One file that you will want to look at is called PACKING.LST. This file contains a short description of the files contained on each of the distribution disks—it's found on the UTILITIES 2 or UTILITIES 1/2 disk. Similarly, you should look to see if a file called README.DOC is on this disk. If it's there, then it contains late breaking information that has not yet found its way to the manuals (misprints and so forth)—it's important to study it.

To get a copy of these files, put a copy of the disk into one of the drives (say, the A drive), make sure the printer is ON and (assuming it's attached to the first printer port) enter:

```
    COPY READTHIS.NOW LPT1:
```

Generally, you can get a hardcopy of any text file by:

    COPY *fname* LPT1:

where *fname* is the name of the file you want to copy (PACKING.LST and README.DOC in this case). This is actually a somewhat crude way to print out a file because it doesn't respect margins or page breaks—you'll see better ways later on.

Another way is to use your favorite word processor. See if it can import text files—almost all word processors can. (Text files may be called ASCII files in the manual of your word processor.) Assuming you can, then it's easy to use your word processor to print these files.

Just what exactly is QuickBASIC 4.5?

A computer's native "language"—usually called machine language—is a seemingly endless series of 0's and 1's. Today, practically no one needs to know which of these sequences do what, because early on in computing people developed programs called assemblers that, at the very least, allowed you to use mnemonics for the basic operations: instead of writing 0001001 you could write ADD.

Assembly language gives you unsurpassed control over your machine—at a price. Each type of computer has its own assembly language, and they are all cumbersome. In the mid-1950s the computer world recognized that they had to have computer languages other than the languages designed for specific machines. As long as a program for one type of machine was useless on another, progress was slow; programs were less useful and programmers less versatile.

This is not to say that one can escape assembly language completely. Most compilers are written at least partially in assembly language.

The first widespread solution to this problem was a scientific language called Fortran (FORmula TRANslation). Programs written in Fortran can run on any computer that has a special program, called a Fortran compiler, written for it. The compiler translates a Fortran program into the appropriate machine language instructions for that machine. Fortran has many features that are useful for scientific applications: it is still widely used on the CRAY supercomputers and for scientific applications in general. However, it was always unpleasant for beginners (or, for that matter, any nonscientist) so in 1964, at Dartmouth college, an offshoot of Fortran called BASIC was designed by John Kemeny and Thomas Kurtz. (BASIC stands for Beginner's All Purpose Symbolic Instruction Code—though the abbreviation might have come first.) BASIC is now the most widely known of all computer languages and among the most versatile. (COBOL was the other early language that survived. It is used almost exclusively for business applications and never caught on in the microcomputer world.) Almost all microcomputers come with a version of BASIC.

Because almost all PC's come with a version of BASIC, why did Microsoft decide to market QuickBASIC for IBM PC's and compatibles? Why were they convinced it would sell? (And sell it does—BASIC is still the most popular language for programming, and QB outsells all other BASIC compilers by around 8 to 1.)

To answer these questions, you need to know something about the differences between compilers and interpreters. First of all, they differ in how they translate not in what they translate. Both start with a series of instructions—usually called the source code—that make up your program. A BASIC interpreter (like the one supplied with most machines) translates the instructions in your source code one line at a time. It processes the translation, and then it goes back for the next instruction until it's finished with your program.

A compiler, on the other hand, translates the whole program into machine language at once. The result—usually called *object code*—can (with only a little extra work) be used independently and repeatedly. (Object code has to be *linked* with certain support routines to give an executable file. Linking is done by a separate program, but in QuickBASIC 4.5 it usually doesn't need intervention on your part. You rarely need to know the details of this process.)

Compilers have two main advantages over interpreters, both of which will give you programs that run faster. The first is that you can save the translation. This way, if you need the program again you don't have to have the machine retranslate it. In particular, you don't need to have a BASIC interpreter or compiler around any more. The linked object code will stand alone.

The second advantage occurs for instructions that are used many times. The interpreter must retranslate the instruction each time it is encountered; the compiler translates it just once. If an instruction is repeated a few hundred times, then the time saved is considerable.

So, you might be wondering why people use interpreters at all? The main advantage is the immediacy that is possible with an interpreter. Type in a command and the computer processes it. Make a mistake and it's easy to change. With a compiler on the other hand, make the slightest change and you have to do the whole translation again. Because the translation process can take a considerable amount of time—even for the simplest program—a simple typo might mean five minutes wasted. Moreover, it might be difficult to isolate exactly where the typo was.

This problem occurs because a compiler usually gives results that are impossible to associate with the original program. Even an expert programmer would be hard pressed to translate compiled code back into your source code.

Microsoft changed all this with QuickBASIC 4.0. QuickBASIC 4.0 should be called an *interpiler*. Think of this as a combination compiler and interpreter. As you write your program the QuickBASIC 4.5 "interpiler" translates your program into an intermediate form. (This intermediate form is called *threaded p-code*—but exactly what *that* is is a bit difficult to explain.) What's important is that there's still a correspondence between the p-code and your original program. The compiler can easily go back and forth between the two, so that you lose none of the immediacy possible with interpreters.

Also, once the program has been transformed into p-code (which happens as you enter each line—the delay is barely noticeable) you can run the program much as you did with traditional interpreters, but the speed is close to that of a traditional compiler. Because you've lost little of the immediacy, you can still stop the program, make changes, and continue without having to re-compile the program.

Finally, once you're completely satisfied, you can compile the program in the traditional way.

Actually, Microsoft made the standard BASIC interpreter supplied with your machine obsolete for another reason: they greatly enhanced BASIC. QuickBASIC 4.0 contained many new commands—and improved versions of old commands (and QB 4.5 has even more). And, although almost any program that ran under the old, interpreted BASIC will still run (probably many times faster) using QuickBASIC, now you can write programs in a more modern, "structured" way.

Structured programming is difficult to define precisely—but the problems it hopes to correct are real. As programs grow longer, the possibility for mistakes grows even faster. One of the complaints that professional programmers had about BASIC is that it encouraged bad programming habits. Over the years computer scientists have discovered ways to make large programming projects less painful, both in time and money. The key idea is that your programs should be organized like the households of the very rich. The very rich don't often do housekeeping, run errands or take care of similar chores. Instead they have various subordinates who take care of these details. Programs are best written with the same type of organization. A powerful and flexible program should be made up of a few controlling statements, together with many subordinate "sub-programs" for the nitty-gritty jobs.

The problem with traditional BASICs like the one that was probably supplied with your machine is that they lack the commands that make this possible. Because the "syntax" of using specific commands is available on line, I intend in this book to concentrate on teaching you these techniques.

Batch files, prompts and CONFIG.SYS

Often you want your computer to *initialize* itself when you turn it on. For example, instead of being greeted by the infamous "A prompt"—A> (or C> if you have a hard disk) you might prefer something like the date, time and full pathname:

```
Mon 1-1-89
9:46:01
Current path is:  C:\QBBOOK\CH1
May I help you?
>
```

This might be overdoing it. Customizing your system is done from a file called AUTOEXEC.BAT that must be on your boot disk (the disk or drive from which DOS loads). After DOS is loaded, DOS immediately looks for a file called AUTOEXEC.BAT and executes any of the DOS commands that it finds there.

The DOS command that changes the prompt is called, naturally enough, PROMPT. Here are some of the options for this command:

PROMPT $D	gives the date
PROMPT $P	gives the current path name
PROMPT $T	gives the current time
PROMPT $_	gives a new line
PROMPT $G	gives the >

You can combine these options: PROMPT D_$P gives the date on one line and the path on a new line. Finally, to get a message just enter the message after a $. The PROMPT command that gives the example above is:

```
PROMPT $D$_$T$_$P$_$ May I help you?$_$G
```

You can try various combinations of the PROMPT command. They go into effect immediately, and the command PROMPT alone goes back to a >.

Another command that you are likely to want in your AUTOEXEC.BAT file is the PATH command that tells DOS where to find programs. For example, this lets you start your word processor or QuickBASIC from any directory—not just the one they're in. Also, you will want commands to load your favorite memory-resident programs. I add a directory to my path called "BATCH," that contains the vast majority of my batch files.

In general, any file with the extension .BAT is used to hold groups of DOS commands that are meant to be executed at one shot. For example, I have a file called QB.BAT in the "BATCH" directory of my hard disk that contains the following two commands:

```
CD \QB45
QB
```

Now when I'm anywhere at all, I enter QB and DOS looks in the BATCH directory (included in the path) and executes both these commands. Thus, I end up in the QB45 directory and running QuickBASIC. Batch files make navigating the vast expanses of modern hard disks possible. In fact, batch files can be programs in themselves. Once you know more about QuickBASIC you might want to learn batch file programming. Consult any good DOS book for this.

Another way to customize DOS is with a file called CONFIG.SYS. This file can partially control how DOS works. For one thing, it helps control how the RAM in your system is allocated. For example, you use CONFIG.SYS to set up a RAM disk. If you have 640K of memory then you'll probably want to make sure that your CONFIG.SYS file contains the commands:

```
BUFFERS = 20
FILES = 20
```

These commands set aside more space for DOS (and QB) to handle files. Many application programs work faster with this type of configuration.

CONFIG.SYS or a .BAT file must be in text (ASCII) form. There are many ways to create such files. You could use the horrible EDLIN supplied with DOS, or a very sophisticated word processor. But, because QuickBASIC comes with a built-in editor, I'll show you how to do this in the next section using QB itself.

There are two more features of DOS that I want to introduce: pipes and filters. These are commands that *redirect* the results of a DOS command somewhere else. For example, suppose you want to get a printed copy of a directory. If the directory only fills one screen you could use Print Screen, but if it is any larger, you're in trouble. The command:

```
DIR > LPT1:
```

redirects (sends) the result of the DIR command to LPT1—the printer that's hopefully attached to the first parallel port. Generally, the > is used to send the result of a DOS command, ordinarily displayed on the screen, somewhere else. For example

```
TYPE PACKING.LST > COM1:
```

sends the contents of the file PACKING.LST to the first serial port and

```
TYPE PACKING.LST > LPT1:
```

is a more complicated version of COPY PACKING.LST LPT1:. Similarly:

```
DIR > "A:DIRFILE"
```

sets up a file containing the contents of the current directory.

Another useful feature is *filtering*. A *filter* (which is usually a small program) takes information, processes it in some way and then sends the result somewhere else. One of the most useful of these filters is the MORE filter. MORE changes the way the display works with long files. Now, after each 23 lines, the display stops scrolling and waits for you to press a key before continuing (much like the DIR/P command). For example:

```
TYPE README.DOS | MORE
```

lets you examine the README file one screen at a time—without having to constantly fight with Ctrl+Num Lock. | is usually located above the backslash key—it's not a colon.

For this to work, of course, DOS has to know where the MORE program is—it's usually found in the same place as DOS itself. Make sure that the PATH includes the location of any filters you use. Other filters let you SORT a file or FIND a file that contains a piece of information.

Obviously, I've only touched on the power of DOS in these two sections. Anyone seriously interested in programming for PC's should know much more, batch file programming in particular.

The QuickBASIC environment

This section shows you how to use the menus that make up the QuickBASIC 4.5 programming environment. When you finish this section you'll also be comfortable with: the "smart" editor; the file manager built into QuickBASIC 4.5; and know something about QB's "on-line help."

Let me point out again that *QB Express*, the on-line instruction program (LEARN.COM) that comes with QB 4.5 nicely complements the material given here.

If your machine has a hard disk then make sure you have set the path to tell DOS where to find QB. Another possibility is to create a batch file like QB.BAT, mentioned in the previous section (you'll see how to use QB's editor to do this shortly). Although it obviously means a bit more typing, the least sophisticated option is to CD (change directory) to the directory holding QB 4.5—if you've used easy setup it's QB45. If your machine is floppy based, put a disk that contains the QB program, QB.EXE, in the active drive. Now enter:

```
QB
```

to get started. (If you have a monochrome monitor attached to a graphics card, you're better off entering QB/B—see appendix A for other ways to start QB.)

Because QuickBASIC 4.5 is a fairly large program, be patient. If you don't have a hard disk, then it will take around 30 seconds to load. The first thing you'll see after it loads is the copyright notice.

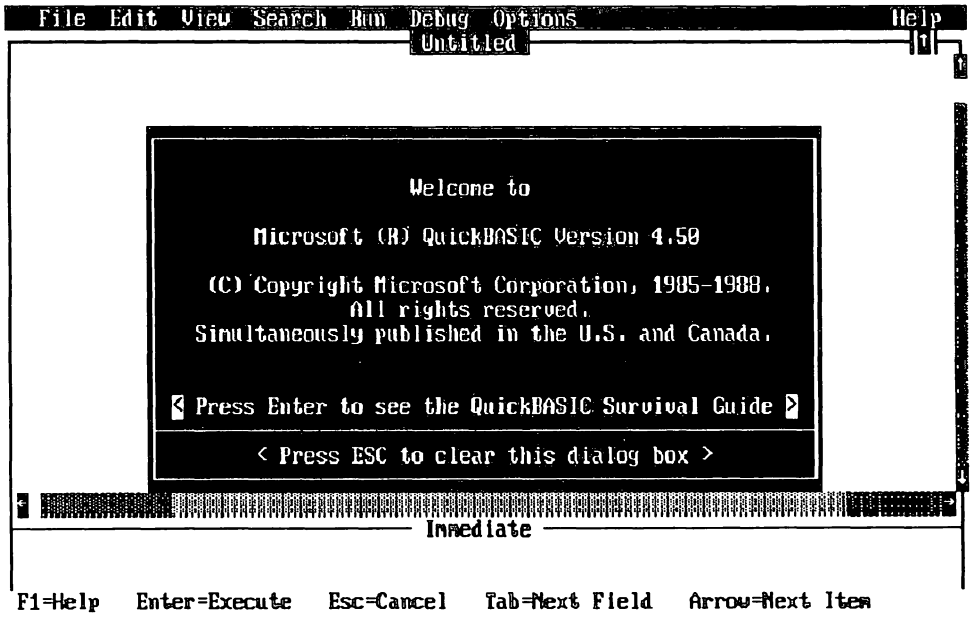
The QuickBASIC survival guide mentioned here is basically a reworked version of QB Express and it starts from the screen shown in Fig. 1-1.

In any case, once you've mastered the material in this chapter or the similar material in QB Express, then you'll only want to hit Esc to go directly to the QuickBASIC main screen (see Fig. 1-2).

Notice how the screen is divided into one large window (called the *View window*) and one thin window on the bottom. The cursor (the blinking _) is in the View window. The cursor marks the *Active window*. The bottom (thin) window is (as indicated) called the *Immediate window* and you'll learn more about it soon.

To get started try the following: simultaneously hold down the Ctrl key and the F10 function key (henceforth, I'll say something like "enter Ctrl+F10"). Notice the View window has expanded and the Immediate window has disappeared. Now enter Ctrl+F10 again; you're back to where you were at the start.

Ctrl+F10 is a *toggle*. It switches the active window back and forth to full size.



Welcome to

Microsoft (R) QuickBASIC Version 4.50

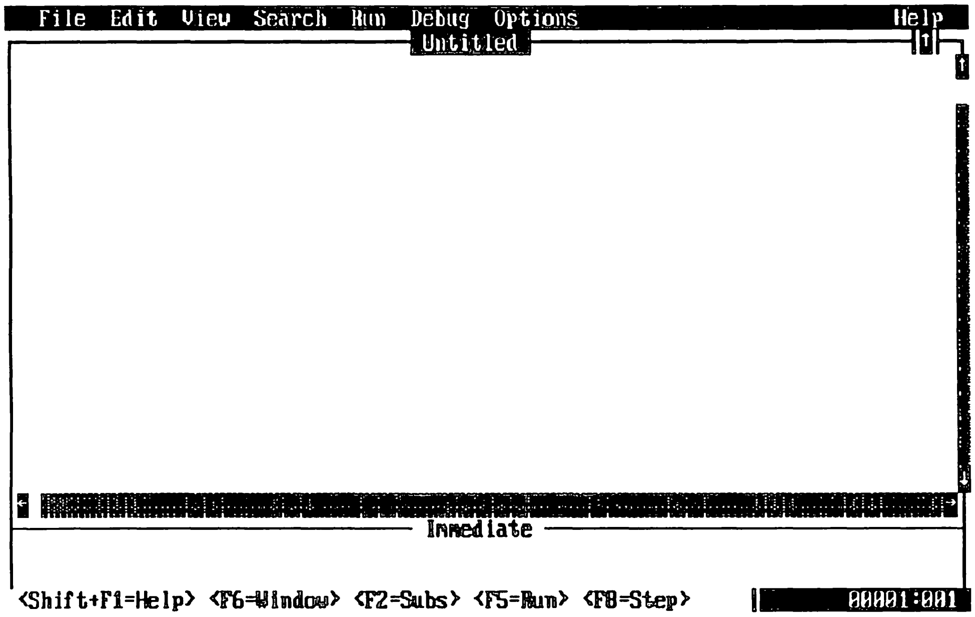
(C) Copyright Microsoft Corporation, 1985-1988.
All rights reserved.
Simultaneously published in the U.S. and Canada.

< Press Enter to see the QuickBASIC Survival Guide >

< Press ESC to clear this dialog box >

F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

1-1 The survival guide.


File Edit View Search Run Debug Options Help
Untitled
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00001:001

1-2 The QB main screen.

To change the active window use F6. Try it now. Notice the cursor is now in the Immediate window. (It's now the active window.) Press Ctrl+F10 again to expand the Immediate window to full size. Bring it back to normal. Finally, use

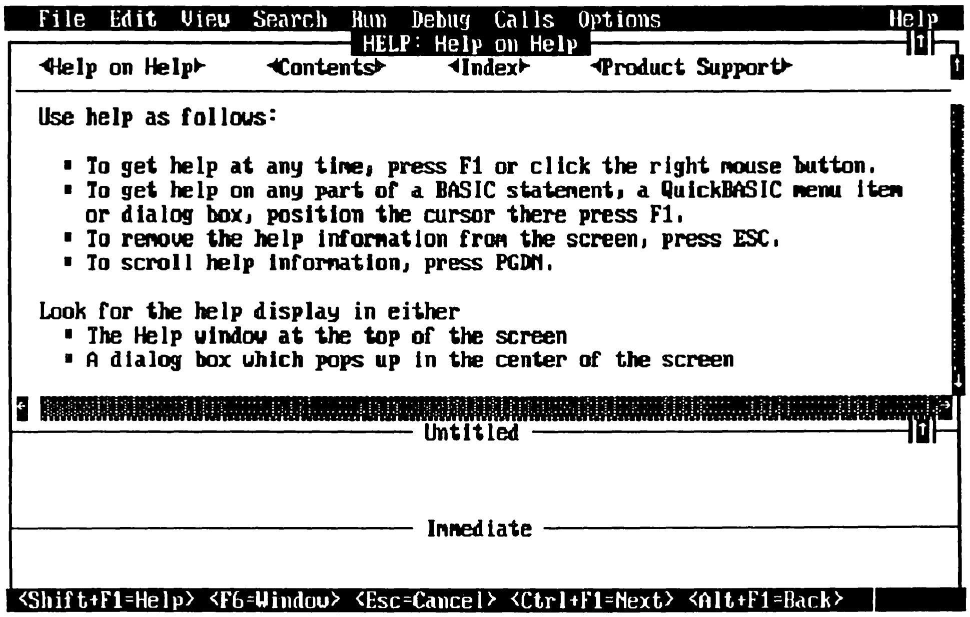
Shift+F6 to make the View window active again. F6 cyclically activates the next window going from top to bottom (and wraps around to the top again). Shift+F6 moves the active window from bottom to top to bottom.

F6 makes the next window active. Shift+F6 makes the previous one active.

You can also shrink or enlarge a window one line at a time. The Alt and the plus (+) key enlarge the active window and the Alt and the minus (−) key shrinks it.

Next, notice the *menu bar* running along the top. The options on the menu bar control the environment of QuickBASIC. The bottom line of the screen (the manual calls this the *reference bar*) tells you some of the current shortcut keys. It also tells you if the caps lock is pressed (a C shows up), the num lock key (an N) and where you are in the file.

Right now it indicates that Shift+F1=Help. So press Shift+F1. Now, if you have a hard disk and QB knows where the help files are (and it will if you've used easy setup) then you'll see a screen like Fig. 1-3. Otherwise, QuickBASIC will prompt you for where the help files are. Place the appropriate disk in one of the drives and give the full pathname of the help file.


File Edit View Search Run Debug Calls Options Help
HELP: Help on Help
◄Help on Help► ◄Contents► ◄Index► ◄Product Support►
Untitled
Immediate

```
Use help as follows:

  • To get help at any time, press F1 or click the right mouse button.
  • To get help on any part of a BASIC statement, a QuickBASIC menu item
    or dialog box, position the cursor there press F1.
  • To remove the help information from the screen, press ESC.
  • To scroll help information, press PGDN.

Look for the help display in either
  • The Help window at the top of the screen
  • A dialog box which pops up in the center of the screen
```

```
<Shift+F1=Help> <F6=Window> <Esc=Cancel> <Ctrl+F1=Next> <Alt+F1=Back>
```

1-3 The initial help screen.

Notice that the cursor is now in this Help window, indicating that this is now the active window. Press Ctrl+F10 to enlarge it to full size. You'll see a lot more about the integrated help system in the next section, but for now notice that the reference bar has changed. The controls found there are context sensitive—what you see (and what you expect to control) depends on where you are. For example, one of the options now is <Esc = Cancel>. This means that pressing the Esc key closes the help window. Press Esc to close this window. When you're working with QuickBASIC the Esc key will almost always bail you out of a situation. Press it a few times and you'll usually end up back in the main screen with the ability to work with the options available on the menu bar.

More on Help

Go back and press SHIFT+F1 and expand the screen with Ctrl+F10. Now that the Help window is the active window, you can scroll through the information in it either with the arrow keys or the PgUp/PgDn keys. (This particular "Help on Help" information is about 2 1/2 screens long.)

The most important thing to know when using QB's online help is what a *hyperlink* is. Hyperlinks are words or phrases that are usually surrounded by highlighted or green angle (on a color monitor) brackets (< >). Think of these as gateways to new information. There are four in this initial "Help on Help" screen:

Help on Help	Which is highlighted because it's currently active.
Contents	Is the "table of contents" for the help files.
Index	Lists all QB keywords.
Product Support	Some of the ways to get help.

As you become more familiar with QB you can use the Index for information about specific commands. It turns out that although they are not surrounded by angle brackets, commands are hyperlinks also.

When the cursor is within a hyperlink, pressing Enter sends you to the information contained at that link. You can move forward to the next hyperlink with the Tab key or back with Shift+Tab. Even simpler is to press the letter of the first word in the hyperlink. This quickly moves to the next hyperlink with that letter as its first character. Case is irrelevant—in fact, Shift+any character moves back to the previous hyperlink with that leading character. For example, if you're in the screen indicated in the figure above, then pressing P moves the cursor to the hyperlink for "Product support." Once you're at a hyperlink press F1 or hit Enter to access that information. Alt+F1 moves to the previous help screen. You can also print out any of the help screens—see the next section.

As I mentioned above, each command is also a hyperlink, so this help screen is the gateway to information about all the commands in QuickBASIC 4.5. Once you hyperlink to the information about a command, then you can get both details and examples about that command. The usefulness of this information can't be stressed enough: it makes the old fashioned (thick!) reference manual obsolete. (And so, in my opinion, not worth the money that Microsoft is charging for it.)

To move within the index you can use PgUp or PgDn or a shortcut: press a letter and the cursor moves to the first command that begins with that letter (as with any other hyperlink).

There is actually another kind of help: environment help. This shows up when you ask for help about dialog boxes, menus and various errors. This information is stored in the file QB45ENER.HLP (on the UTILITIES 2 disk). The information that you've just sampled is contained in two files: QB45QCK.HLP ("quick help" on the program disk) and QB45ADVR.HLP (on the QB advisor disk).

Here's a summary of the keys to manipulate the help files:

Shift+F1	Gives "Help on Help."
Ctrl+F1	Go to help on the next topic.
Shift+Ctrl+F1	Go to the previous topic.
Alt+F1	Traces back through 20 previous hyperlinks.
Tab	Move to the next hyperlink.
Shift+Tab	Move to the previous hyperlink.
up arrow key	Moves cursor one line up if possible.
down arrow key	Moves it one line down if possible.
PgUp	Moves one screen up if possible.
PgDown	Moves one screen forward if possible.
Press a character	Moves to next hyperlink with that as its leading character.
Shift+Character	Moves to the previous hyperlink with that as its leading character.
Escape	Esc closes the help window.

In addition, if the Help window is active, then you can use the Search menu to Find a piece of information in it or place a "marker" (Ctrl+Q+*n*, n = 0,1,2,3). Once a marker is placed, then no matter where you are in the environment, if you hit the appropriate "move to marker" (Ctrl+Q+n) you open the Help window and end up back in the marked spot. See the last section for more on these features of the editor. (Of course, there's always a mouse—see appendix E.)

Finally, as you become more familiar with QB, then you'll want to study the help file on Syntax Notation. This is a hyperlink available via the Contents screen that describes the conventions and abbreviations used by the help files when explaining commands.

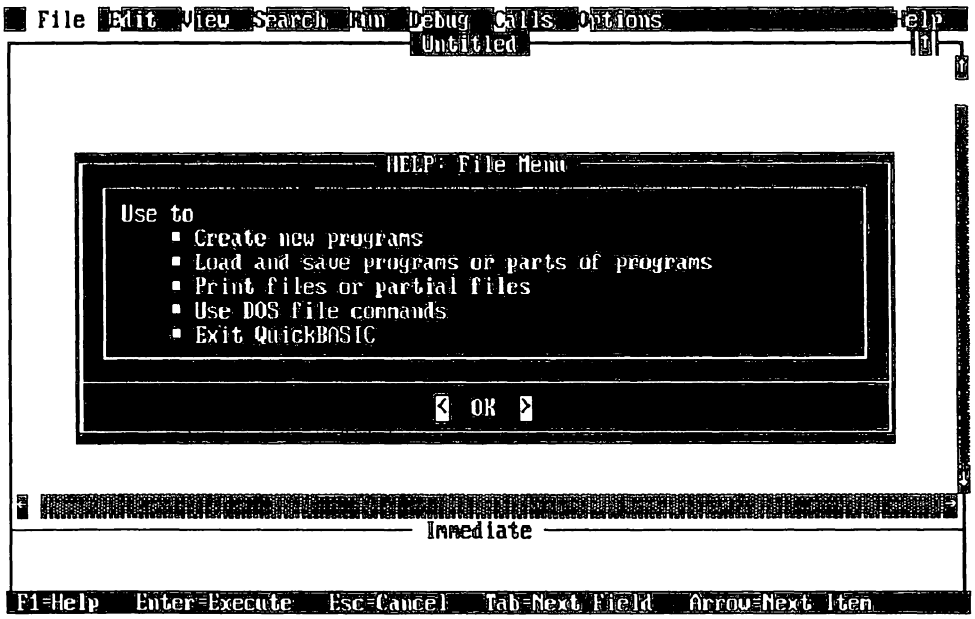
Menus

In this section you'll see some of the power unleashed from the main menu bar. Press the Alt key for a moment. Notice that a bar in reverse video covers the File option and one character on each entry in the menu bar is now highlighted. To open a menu, move that bar around with the arrow keys and hit Enter or press the highlighted letter. As a shortcut, you can simultaneously press the Alt key plus the highlighted letter—for example, Alt+F for the file menu or Alt+E for the Edit menu. If a menu is open, then the right and left arrow keys close that menu and open the one to the right or left respectively.

The File menu controls saving, loading and printing files. It's also how you leave QB (X is the shortcut). Press Alt to make the menu bar active. Notice the File option is highlighted. Before hitting the Enter key to pull the File menu down, hit F1 (the "Help on" key). Notice that a help window opens with information about the File menu that looks like Fig. 1-4.

Although the screen in Fig. 1-4 isn't terribly informative, it is a good example of context sensitive help. Hit Enter or Esc to close this screen.

In general, press F1 (rather than Shift+F1, which accesses the more general "Help on Help") and QB will try to provide help about the choice, option or symbol that the cursor is on.



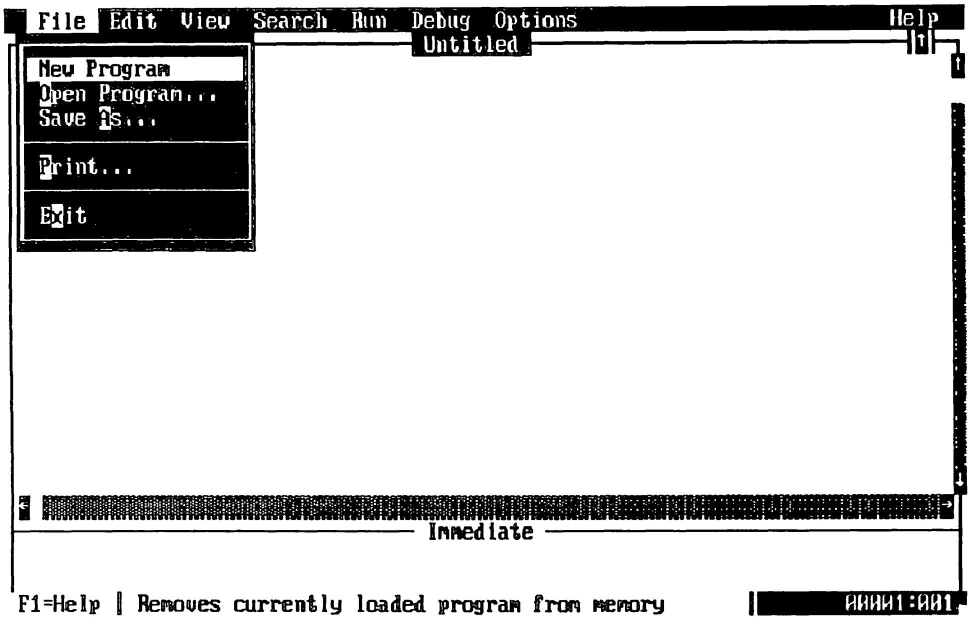
```
Use to
  ■ Create new programs
  ■ Load and save programs or parts of programs
  ■ Print files or partial files
  ■ Use DOS file commands
  ■ Exit QuickBASIC
```

< OK >
Immediate
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

1-4 The help screen for the file menu.

When you open the file menu, you'll see Fig. 1-5 (the shortcut is Alt+F).

You choose options from a menu in either of two ways. The first is to move the highlighted bar around with the arrow keys. Once you're satisfied that you're at the place that you want to be, hitting Enter accepts that choice; or notice that each of the options in a menu has a highlighted letter. Pressing this letter is a shortcut to activate the option. If three dots follow an option (like the one for



F1=Help | Removes currently loaded program from memory 00001:001

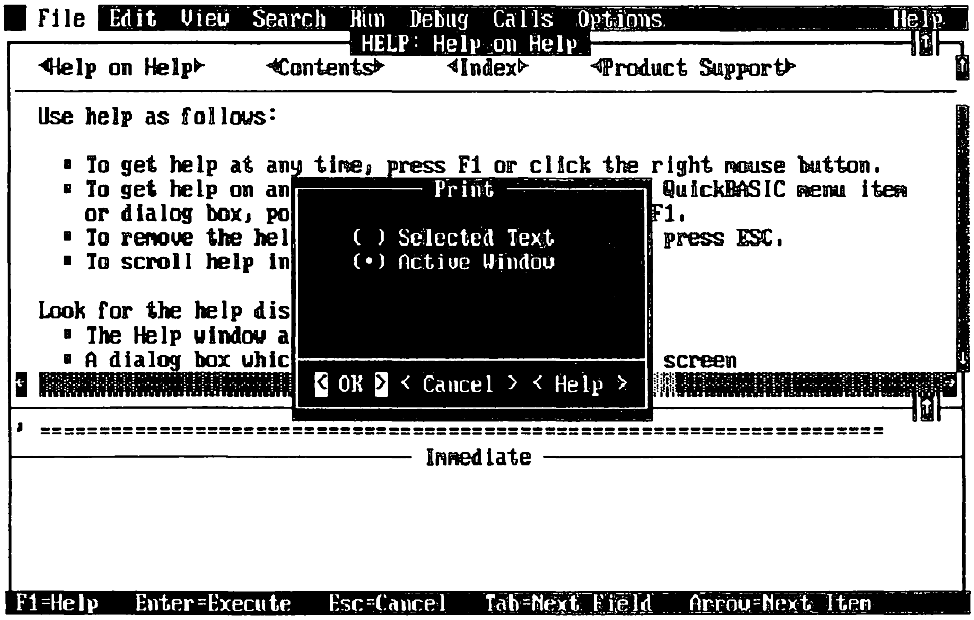
1-5 The "easy" file menu.

Print) this means that, if you choose this option (in our case by pressing a P), then you'll have a *dialog box* to deal with—more on those shortly. Some menu choices have a third way—a super shortcut key that takes you directly from the View or Immediate window. For example, you already saw that Help on Help is available while in the View window through Shift+F1.

If you have a printer, then it's easy to get a printed version of any help screen. (In fact, printed versions of all the help screens are essentially what Microsoft is selling.) To do this use Print on the file menu. Hit Alt+F to open this menu and P for the print option. The dialog box in Fig. 1-6 pops up.

Now all you have to do is make sure that the help screen that you are interested in is the active window. Then, assuming your printer is on line, hit Enter to get a hard copy. Of course, this only works if the bullet is in the option marked active window. The arrow key moves the bullet. You also can select a portion of the information and print that piece; or, via the Edit menu, copy it and paste it into your programs. More on this in the last section of this chapter.

Now try to run one of the sample programs that came with QuickBASIC 4.5. Open the File menu and notice that the New Program option is highlighted. This means that if you now hit F1, you'll get help about the New Program command. In this case we want to Open a program. So hit O (the highlighted letter). What pops up is a dialog box that looks like Fig. 1-7.

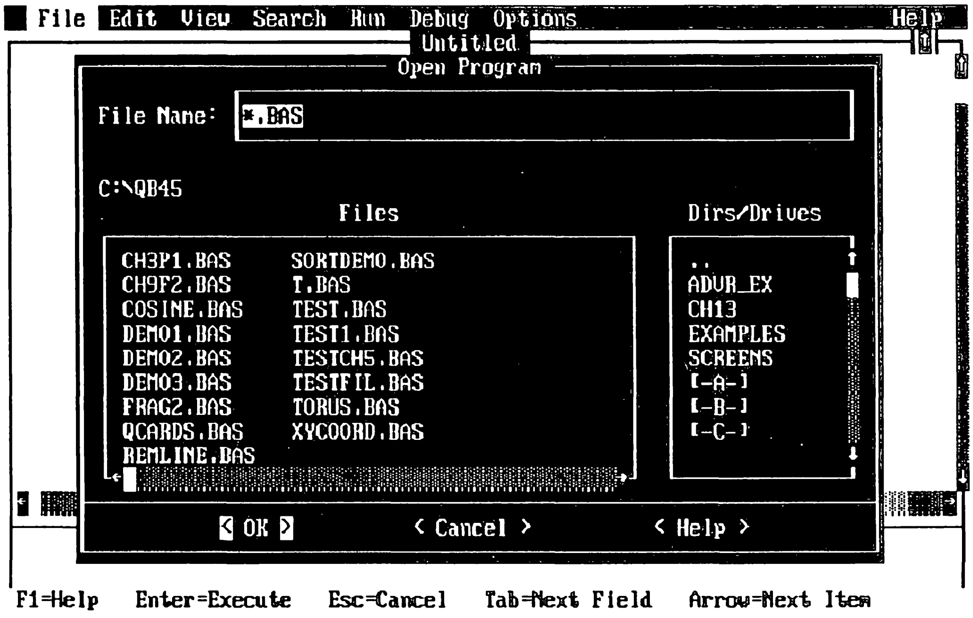

File Edit View Search Run Debug Calls Options Help
HELP: Help on Help
◄Help on Help► ◄Contents► ◄Index► ◄Product Support►
Print
( ) Selected Text
(•) Active Window
< OK > < Cancel > < Help >
Immediate
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

Use help as follows:

- To get help at any time, press F1 or click the right mouse button.
- To get help on an ... QuickBASIC menu item or dialog box, po... F1.
- To remove the hel... press ESC.
- To scroll help in...

Look for the help dis...
- The Help window a...
- A dialog box whic... screen

1-6 The print dialog box.


File Edit View Search Run Debug Options Help
Untitled
Open Program
File Name: *.BAS
C:\QB45
Files
Dirs/Drives
< OK > < Cancel > < Help >
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

```
CH3P1.BAS     SORTDEMO.BAS        ..
CH9F2.BAS     T.BAS               ADVR_EX
COSINE.BAS    TEST.BAS            CH13
DEMO1.BAS     TEST1.BAS           EXAMPLES
DEMO2.BAS     TESTCH5.BAS         SCREENS
DEMO3.BAS     TESTFIL.BAS         [-A-]
FRAG2.BAS     TORUS.BAS           [-B-]
QCARDS.BAS    XYCOORD.BAS         [-C-]
REMLINE.BAS
```

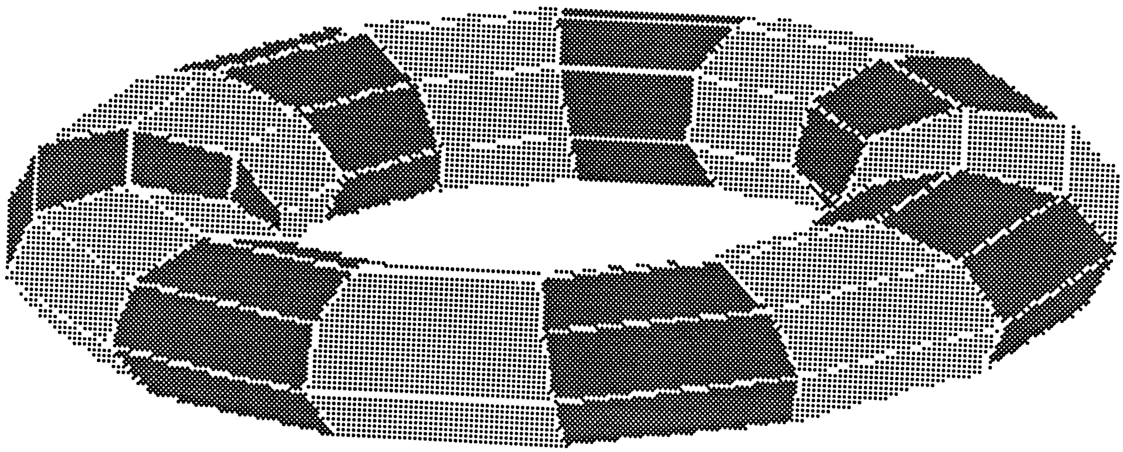
1-7 The dialog box for Open program.

Dialog boxes require a bit of getting used to, but for now notice that you can move the cursor back and forth with the Tab (forward) or Shift+Tab (backward). Where the cursor is within a dialog box is called the *input focus*. What you type appears there.

I want to open (load) one of the sample programs, TORUS.BAS, that came with QB. This is a pretty neat program if you have a graphics card—the display is quite dramatic. It comes on the SETUP disk but, if you used easy setup to load QB onto a hard disk, it is now in the QB45 directory. Make sure the cursor (input focus) is in the box marked "File name" and type TORUS.BAS. Now hit Enter to accept this choice. The drive will whirl as QB loads this program.

To run it press Shift+F5 and hit Enter to accept the built-in choices. When the program is finished press Esc twice to go back to the QB environment. The screen dump in Fig. 1-8 doesn't do the colors justice, but for those who don't have a graphics card, Fig. 1-8 is what this program displays.

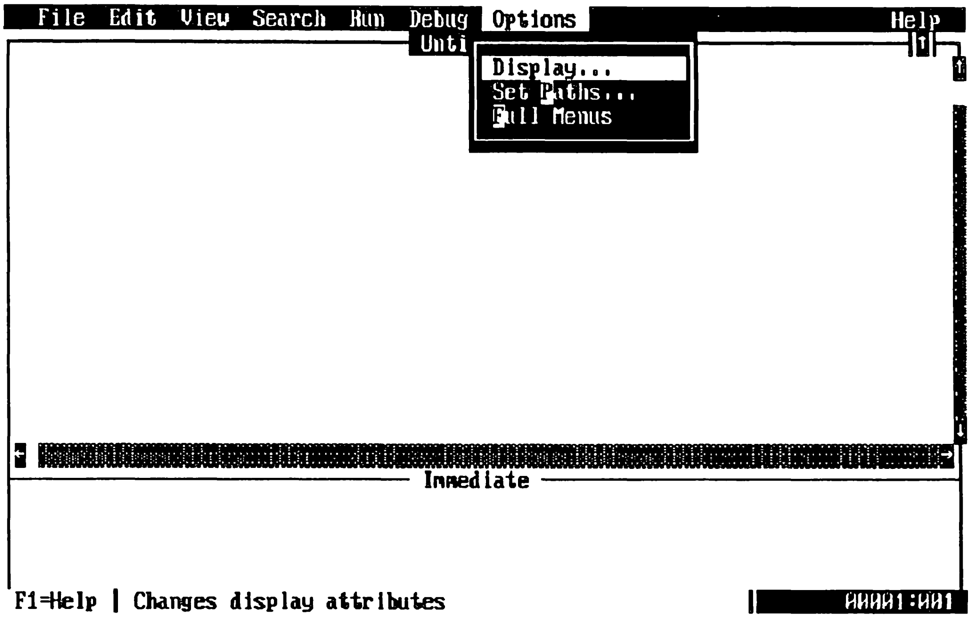
If QB tells you it can't find the program, press Enter to close the error box and choose the Open Program option again. Only this time give the full path name of TORUS.BAS. For example, place a copy of the setup floppy disk in drive A, open the program dialog box, and this time enter A:\TORUS.BAS.


1-8 The torus demo.

More on menus

Suppose now that you press Alt+O to open the options menu. If you try it, then you'll see Fig. 1-9.

Notice that the first option is called Display . . . this lets you customize the display—what colors are used for highlighted text and so forth. The next option is called Paths. This holds the paths (see the previous section) where QuickBASIC can find the information it needs. (QB always checks the current directory as


File Edit View Search Run Debug Options Help
Unti
Display...
Set Paths...
Full Menus
Immediate
F1=Help | Changes display attributes
00001:001

1-9 The Options menu.

well.) If you have a hard disk, and used *easy setup*, then this information is already set. If you move any of QB's essential files, or if you have a floppy based system, then you might have to change this information. This information is stored in a file called QB.INI that is updated whenever you leave QB. This in turn means that changes to this information can be made any time you are in the QB environment. You don't need to run the SETUP program again.

Next, notice the last option. It's called Full menus. QuickBASIC comes with two types of menus, usually called Easy menus and Full menus. Easy menus have fewer options and a bit less power. Full menus, on the other hand, have almost an embarrassment of riches. I'll need some of the features of Full menus in this chapter, but on the whole, I only toggle them ON when I really need them.

Back to Full menus—if there's a bullet next to it on the menu, then Full menus are already ON. If not, press F. Notice the option menu closes and the top menu bar has one more option (the Calls menu) that I won't use until chapter 8.

Open the option menu again (Alt+O). Notice that there are a few more choices than before and there is a bullet next to Full menus. The bullet indicates that this (or any option) is toggled ON. Next, notice one of the options in this menu is called Syntax checking. While you're learning the editor, you might want to turn Syntax checking off. If there's a bullet there, then toggle it off by pressing S, or move the highlighted bar to it and press Enter.

Turn Syntax checking on and you've activated the smart editor. Like many programming editors it "knows" some of the rules for QB, and it won't let you alone if you try to break them. Because some of the things you might practice typing, like batch files, will surely break QB's programming rules, for now you should leave Syntax checking off. When you start programming (for example, by working through the next chapter), then you'll want what you type to be checked. So when you've finished working through this chapter, I suggest turning Syntax checking back on; and then go back to easy menus.

The anatomy of a dialog box

QuickBASIC pops up a dialog box whenever it needs information to complete a choice. This situation is usually indicated by three dots on a menu option. The dialog box for Open program demonstrates most of their features. First, it has a place to enter information—a *text box*. Next, there's a list of files and a list of directories or drives—two *list boxes*. Finally, there's a bunch of choices at the bottom. Each of these choices is surrounded by angle brackets (< >). These are called *command buttons*, and one command button is always surrounded by highlighted angle brackets. This is the active button, and if you press the Enter key, then the dialog box closes and you've chosen that command. The command in turn uses whatever is listed at the moment in the various text boxes for its raw data. It is important to remember that you do not move around a dialog box with the Enter key.

How do you move around a dialog box then? You've seen one way: the Tab or Shift+Tab key. Suppose you wanted to get to the command button for Cancel to close this box (which is actually silly because you can just press Escape, but bear with me). You could hit Tab four times or Shift+Tab twice. If you do, then you'll notice that the Cancel button is now active (it's surrounded by highlighted angle brackets).

However, for moving to most places within a dialog box, there's a shortcut like the one for menus. For example, simultaneously pressing Alt+H moves the cursor (input focus) directly to the Help button and makes it active. In general, you can reach most places in a dialog box with an Alt+ a letter press—look for a highlighted letter. List boxes are usually used to display information needed to fill text boxes. For example, go back to the Open program dialog box and hit Alt+F to quickly move the cursor (input focus) to the Files list box. Move through this list box with the arrow keys. Notice that as you do so a choice is highlighted and automatically placed in the text box. Once you're satisfied with the choice in the text box, then you hit Enter to open that program. This is because the name of the program was transferred into the text box—you don't have to do any typing at all.

There's even a shortcut for moving within a list box—it works like the one for moving from hyperlink to hyperlink. Once the cursor is in a list box, then if you press a letter the highlighted bar immediately goes to the first item on the list having that first letter. If you press Alt+F and then T, you'd be much closer to loading the TORUS.BAS file.

The Dirs/Drives gives you a list of possible drives and directories. If you make a choice here and hit Enter then the dialog box doesn't close. Instead, you'll be faced with a list of programs in the new directory to choose from.

One final point about the Open program box. Normally it displays a list of files that have the .BAS extension. You can change this by moving the cursor here and typing whatever wild cards you want. For example, press Alt+N and enter *.*, and you'll see a list of all the files in the current drive or directory.

The Edit and Search menus

QuickBASIC 4.5 comes with a full screen editor that can insert, delete, find and replace. Those people who have used WordStar or any of Microsoft's word processors will find many of the commands familiar. In this section I'll show you only the most important commands. A full list of editing keys is available on line (follow the hyperlinks from the Contents in the "Help on Help" box) or go to appendix D.

A full screen "programmer's" editor is used much like any word processor. The main differences are that a programming editor often lacks features like word wrap and print formatting, but it should (and QB's smart editor does) have powerful features that make programming easier.

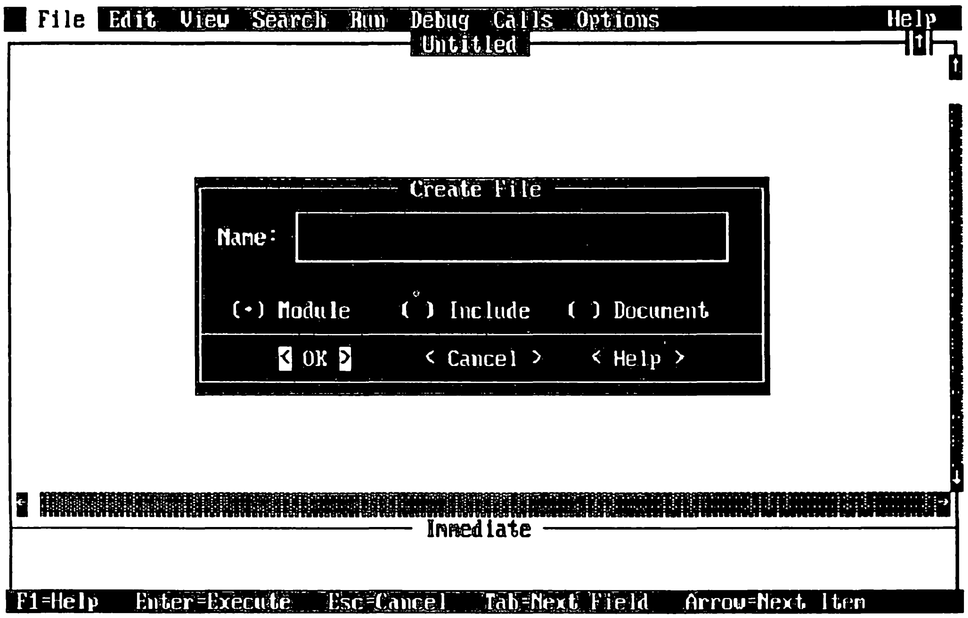
As with any editor, you have two choices when typing: you can either be in *insert mode* or *typeover mode*. When you are in insert mode (the default) any text you type "pushes" the text that follows aside. In typeover mode, the text replaces the old text one character at a time. You switch between these modes by hitting the insert key. When you are in typeover mode the cursor looks like a box. Also in both modes you have the usual two ways of deleting a single character: the Backspace key deletes the character to the left of the cursor, and the Delete key does it to the character at the cursor.

As a way to become comfortable with the editor, let's make the QB.BAT file mentioned earlier. For this, open the (full) file menu and hit C (for Create). Figure 1-10 shows the dialog box that opens up.

First off, you're creating a document, which is what QB calls a "text" file. So press Alt+D to choose this option. Open a file as a document and Syntax checking is automatically turned off.

Now, assuming that you have a hard disk and it's the C drive, and you've set up a sub-directory called "BATCH," move the cursor back to the text box and type C:\batch\QB.BAT. Hit Enter to accept this choice. The dialog box closes and the cursor is back in the view window. Notice that the View window has QB.BAT as its name (instead of "UNTITLED"). Now type CD\QB45, hit Enter and then type QB. The screen looks like this:

```
CD \QB45
QB_
```


File Edit View Search Run Debug Calls Options Help
Untitled
Create File
Name:
(•) Module ( ) Include ( ) Document
< OK > < Cancel > < Help >
Immediate
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

1-10 The create dialog box.

Open the file menu again and hit S for save. The batch file is now on the "BATCH" sub-directory of your hard disk. (Admittedly, using QuickBASIC 4.5 to create batch files is a bit of overkill.)

To follow the next discussion, you'll need to load the file called PACK ING.LST. Open the full file menu again and notice that one of the options is Load File. Press L to open the Load file dialog box. It looks much like the Open programs dialog box, except there's a new row of options (they're called *option buttons*) that look like this:

Load as: (•) Module ( ) Include ( ) Document

The bullet in the Module choice indicates that this is the active option. Because PACKING.LST is a document, we want to turn the last option on instead. Press Alt+D twice to move the cursor to this button and press the spacebar. (This is the general method for toggling options on and off: move the cursor and hit the spacebar.)

Now put a copy of the Utilities 2 disk in drive A, move the cursor back to the File Name (Alt+N) and type A:\PACKING.LST. Now hit Enter to accept the choices. It takes a few seconds for the packing list file to load.

Notice that the cursor is in the view window which is now named PACKING.LST instead of QB.BAT. The first thing we want to do is print this file. Open the File menu and hit P. A new dialog box opens. Make sure that a bullet is in the All modules option. Now, assuming that you have a printer attached to the first parallel port (LPT1:), hit Enter to print a copy of this file.

This file will print but it won't be nicely formatted. It will lack top or bottom margins and too many lines appear on each page. The question obviously is: is there any way to fix this within QB? (In fact, you'll eventually learn how to write your own print formatter, but that's not what I mean.)

The answer is really no: use a word processor as I suggested in the last chapter or a memory-resident program as mentioned in the Introduction; but as a method to teach you the editor, I'll show you a truly horrible way to do so.

First off, notice that in the far right corner of the reference bar are two counters that currently read:

```
00001:001
```

These counters mark the line and column. Because the file was just loaded, I'm at line 1 column 1. We want to put a few blank lines for the top margin. Because the cursor is at the top left hand corner, hit Enter 4 times. Notice that the text moves (scrolls) down 4 lines and the counter now reads: 0005:001.

Hit the PgDn key and notice that you've moved to line 23. (PgUp and PgDn move the width of the window each time—19 lines in this case.) Hit PgDn twice more and use the down arrow key three times to move the cursor to line 62. (This happens to be a blank line in my PACKING.LST.) Now usually each page on a printer is 66 lines, so you'll want to enter at least 8 carriage returns (4 for the top of the next page). Continue this process and you've "hand-formatted" the text.

Anyway, the point of this was to show you that you move through a file in the view window using the arrow keys and PgUp/PgDn keys—just as you did for help files. And, just as for help screens, you also can move to the top (Ctrl+Home) and the end (Ctrl+End) of a document. (The Home and End keys move to the first character and the last character in a line.)

There's another clumsy way to print this file that shows off something more useful: selecting text. When you want to do something with a block of text like move it, copy it or print it, you first have to select it. This is done by first moving the cursor to the start of the block. Now hold down the Shift key while using a mover key (like an arrow key or PgUp/PgDn). As you do so, the text is highlighted. Be very careful after you've selected text:

When text is selected, anything you type (even hitting the spacebar) will replace the selected text.

This means that it's very easy to do something disastrous when text is selected. Select a page, hit the spacebar by mistake, and the page is gone only to be replaced by a space! You must be careful to "de-select" text when you're finished doing whatever you've wanted. This is done by hitting any direction key when you are not pressing the Shift key. (To repeat: hit an arrow key to deselect—never, ever hit the spacebar.)

However, if you're careful, then selecting text is extremely useful. One thing you can do with selected text is print it. For example, suppose you just wanted a copy of the contents of the Utilities 2 disk. Rather than scroll until you find this disk, use the Search menu. Press Alt+S to open and hit F for the Find option. A dialog box opens up that asks you questions. In this case, all you need do is type Utilities 2, press Enter and the cursor moves to where you want to be. Select the contents of this and PRINT it as before. (One other nice feature of QB's search facility is that, when the Help window is active, you can use it to "browse" through the help information. It's a nice complement to the built-in hyperlinks.)

You also use this menu to change text. For example, you could change all occurrences of QuickBASIC to QB. The Change option opens a dialog box that has two text boxes. The first finds the text to be changed and the second is for what to change it to. (Leave this line blank and you can delete any occurrence of the text in the first box.) Notice as well that one option (the default one) is <Find and Verify>, the second says <Change all>. I almost always use the first, which asks, after each "find," whether I want to change that occurrence. Activate <Change all> and it can happen too fast with no going back.

Pressing the Escape key aborts a change and closes the dialog box but doesn't bring back the original version. However there is one kind of "undo." When you make changes to a line, before you move the cursor off the line, hitting Alt+Backspace (=Undo on the Edit menu) brings the line back to its original state. This is very useful—especially if you're prone to over-editing. Unfortunately, once you move the cursor off the line, you no longer can "Undo."

Cutting and pasting is done from the Edit menu. This is always a two-step process: first you select the text to copy, or move. Next, open the Edit menu and choose whether you're cutting or Copying. Make the appropriate choice and QB copies the selected text into a temporary storage area called the *clipboard*. Once information is in the clipboard it can be pasted anywhere in your document. Move the cursor where you want and use Paste (shortcut key is Shift+Insert). A copy of the text that was in the clipboard appears at the location of the cursor. Moreover, because text remains in the clipboard until replaced by a new "cut" or "copy," you can paste multiple copies of the same information.

There are many shortcut keys that you'll pick up over time—cutting, copying and pasting are, for example, Shift+Del, Ctrl+Ins, and Shift+Insert. (To be honest, I usually just open the Edit menu. For example, I find it easier, less confusing and not much slower to use Alt+E and then T for CuT.) The one I find myself using is the "cut a line to clipboard." If you hit Ctrl+Y then whatever line the cursor was on is moved to the clipboard. Now, to get multiple copies of a line in the same place, I just use Paste (Shift+Insert).

Finally, let me remind you that most editing functions have WordStar equivalents, and that all the editing key information is available on-line and in appendix D.

2
CHAPTER

Basic QuickBASIC

The core of QuickBASIC remains the interpreted (GW, BASICA) commands that are familiar to hundred of thousands of MS/PC DOS users. Statements like PRINT and LPRINT work the same. This is why essentially any program written in these (now obsolete) forms of BASIC can run in the QuickBASIC environment (of course, many times faster). The differences come in how you write programs. There's no need for line numbers and the extensions to the control structures will make your programming jobs much more pleasant.

This chapter reviews the "basics of QuickBASIC," but concentrates on the extensions to the control structures (like the "block if-then") and the new form of loops. If you are an expert in an earlier version of QuickBASIC or in GW-BASIC, I hope it will make the transition to the advanced topics covered in the later chapters possible. If I use a command that you're not familiar with in one of the example programs, check the online help for details on it.

First steps

As with any BASIC, a QuickBASIC program consists of individual statements. Each statement must follow the rules of the language. You can't misspell command words like PRINT and expect your program to run. Your task is made easier, however, by the "smart editor" that's built into the QB environment. This editor catches many common typos.

To start writing a program, first make the Edit window active, i.e., make sure the cursor is there (for example, via the New program option on the File menu or pressing the F6 window shifter key enough times). You hit the Enter key after each line of your program.

For example look at this listing:

```
REM      CH2\P1.BAS
REM  a simple print/beep demo

CLS
 BEEP
 PRINT "HELLO WORLD!"

END
```

As you can see, unlike earlier BASICs—like BASICA or GWBASIC—line numbers are not needed in a QuickBASIC program. For QuickBASIC, the end of a program line (sometimes called a *logical line*) comes when you hit the Enter key. QuickBASIC allows you to have very long logical lines—255 characters.

In this book, the first remark statement always contains a name for the program—as a DOS (hierarchical) file name. In addition to being a reasonable way to give cross references within this book, it's also how the programs are stored on the optional companion disk.

The .BAS extension fits QuickBASIC's preconceptions that the source code for any QuickBASIC program has this extension. The load option on the File menu first tries to list only those files having a .BAS extension—although you can override this (see chapter 1).

REMark statements are not processed by the compiler and therefore do not take up any room in your compiled code. This situation is much superior to that of older interpreted BASICs where everything took up space and there was a temptation to skimp on REMark statements as a result. It's easy to question why REMark statements are important—until you try to modify (or even understand!) a complicated program that you wrote months ago.

You can just use a single quote for a remark. This is usually found below the double quote. (It's not the apostrophe that is usually found below the tilde ~.)

```
'  This is also a way to give a remark
```

As with interpreted BASIC, the END command is not, strictly speaking, necessary (when there are no more statements left to process, a program ends), but it's good programming practice. Processing an END command stops the program—even if there are statements that follow it. Once a program ENDs that you are running from the QuickBASIC environment, you'll see "Press any key to continue" on the bottom line of the output screen. Hitting any key now gets you back to the development environment.

To get this program to work—to run it—use the Run option on the main menu bar. One way to do this is to hit the Alt+R combination to return to the Run main menu bar. Once you hit S to run (start) the program, you'll hear a beep and the message HELLO WORLD! is displayed in the output screen.

To save a finished program use the file menu, so use Alt+F to make this menu drop down and press A to activate the Save As option. Because QuickBASIC doesn't yet know the name of this program, a dialogue box pops up. There's a line for the name so enter a file name. You won't be able to use mine unless you have created a directory called CH2 first. (You'll see a way to create directories from within QuickBASIC in section 2 of this chapter.)

Notice that there's another option in this dialog box: Format. The first option (the default) is "QuickBASIC—Fast Load and save." This will save your program in a form that only QuickBASIC can read, and do it in a way that makes moving it in and out of the QuickBASIC environment fast. I prefer the second option so that I can read my source code from other programs, like word processors. You can change to this option before you hit the Enter key by moving the input focus via the Tab key until the cursor is in the Format part of the dialogue box. Now use an arrow key to move the bullet down. The shortcut for this is Alt+T.

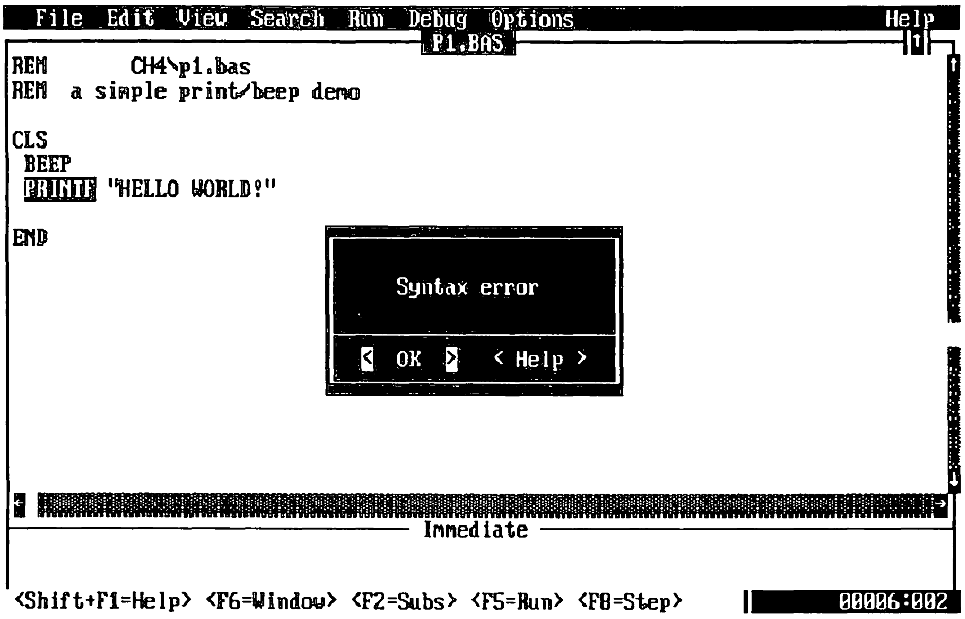
This program had no problems—it compiled and ran successfully. Let's see what would happen if I had made a typo. Make the Edit window active and use the editing facilities of QuickBASIC to force a typo. Instead of the correct command PRINT, make a mistake and have it read "PRINTF." So the changed program looks like:

```
REM         CH2\P2.BAS
REM        a simple typo

CLS
 BEEP
 PRINTF "HELLO WORLD!"

END
```

Now try to run the program. Nothing happens in the output window and all you see is Fig. 2-1 with the misspelled command highlighted.

Now hit Enter and QuickBASIC puts the cursor on the line where the typo was. If you now fix the error using the editing commands described in chapter 1, re-compile the program. It will, of course, work again.

What you just saw was an example of a *compile time* error. This type of error means that your program contains a statement that QuickBASIC can't translate into machine code. When this happens, QuickBASIC stops compiling the program, tells you via a dialogue box what it thinks happened and after you hit the Enter key puts the cursor where it thinks the error occurred.



```
REM      CH4\p1.bas
REM  a simple print/beep demo

CLS
 BEEP
 PRINT "HELLO WORLD!"

END
```

Syntax error

< OK > < Help >

<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step> 00006:002

2-1 Syntax Error dialog box.

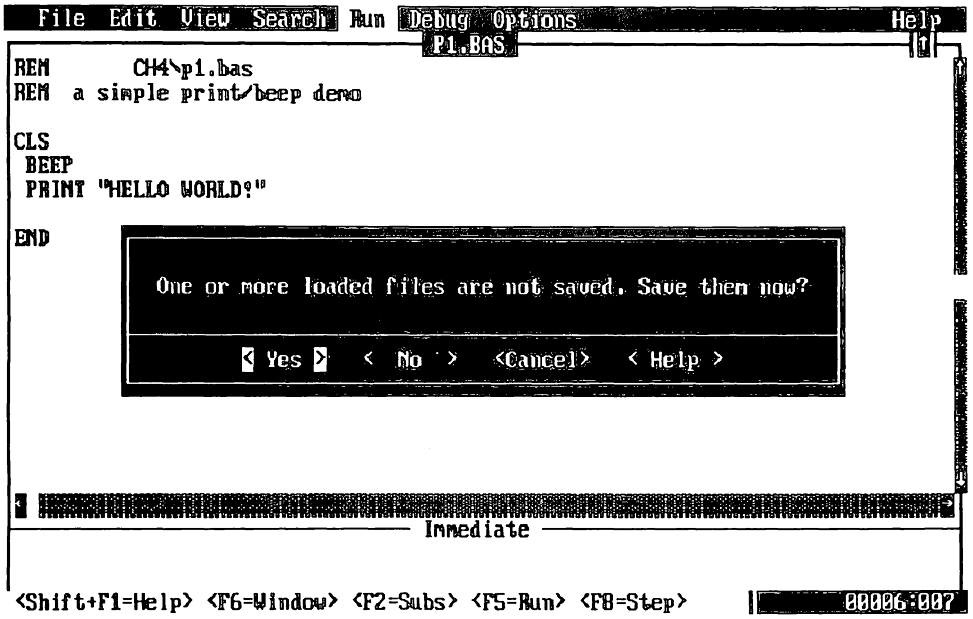
There are also *run-time* errors. These happen when your program is syntactically correct (computerese for having the right "grammar" for a QuickBASIC program) but something forbidden happens while the program runs—a program that tries to divide by 0 for example.

Microsoft says that QuickBASIC has a "Smart Editor," and what this means is that the editor seems to know certain facts about QuickBASIC. For example, I mentioned that QuickBASIC doesn't care if you enter Print, print or PRINT. In fact, as I said then, if you tried any forms of this command but the latter you'll notice that after you hit Enter, this key word is transformed to PRINT. The Smart Editor automatically capitalizes all command words (key words). Thus you can quickly find a typo by checking if what you thought was a key word becomes capitalized. If it doesn't end up in caps you've misspelled it!

You can also compile a QuickBASIC program so that it can *stand alone* (executable from the DOS prompt). For this, use the Run menu. Go to it by the Alt+R shortcut. Notice the last item says:

Make EXE File ...

Make sure this is highlighted and then hit Enter. (You can also hit an X because the X is highlighted.) If you haven't yet saved this version of this program, then a dialogue box like Fig. 2-2 springs up telling you that one or more files have not been saved. This is to allow you to save the *source code* (computerese for the QuickBASIC statements that make up a program). You can't make an EXE file until you save the source code.



File Edit View Search Run Debug Options Help
P1.BAS

```
REM       CH4\p1.bas
REM  a simple print/beep demo

CLS
 BEEP
 PRINT "HELLO WORLD!"

END
```

One or more loaded files are not saved. Save them now?

< Yes > < No > <Cancel> < Help >

Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
00006:007

2-2 A warning box.

After going through the steps necessary to save the source code, you'll get to the dialogue box shown in Fig. 2-3.

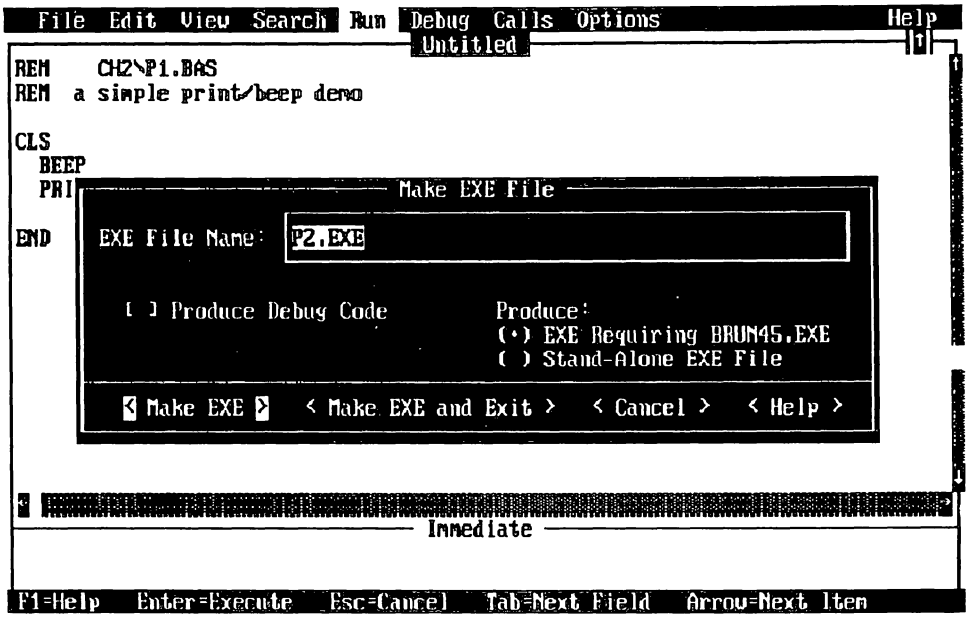
The first asks for a name (P2.EXE is the suggested one). It's a good idea to follow a consistent pattern. Use the extension .BAS for source code and merely change the extension to the required .EXE for the executable version. The option of "Producing debug code" is useful—without it you cannot stop a program by the Ctrl+Break combination. (It has other features that I'll talk about later.) To make the program truly stand alone, you'll have to toggle on the "standalone EXE file" option. Otherwise, the program will require the file called BRUN45.EXE to work.

If you are satisfied with the name, hit Enter to start the process. You also have the option of Make EXE and EXIT (QuickBasic) or the ubiquitous Cancel (Esc). In any case, making an EXE file requires temporarily exiting QuickBASIC and takes some time.

If you have a hard disk and you've installed QuickBASIC correctly, everything happens automatically and all you'll notice is the appropriate disk drive whirl as QuickBASIC sends the standalone version of the program to the disk. If you're using a floppy based system, or haven't told QuickBASIC where to find certain utility files, dialogue boxes will be prepared for multiple disk swaps. You'll need the following three files that are found on the utilities disk(s):

BC.EXE, LINK.EXE and BCOM45.LIB

as well as the main QuickBASIC program QB.EXE. When you're asked for one



```
REM   CH2\P1.BAS
REM  a simple print/beep demo

CLS
  BEEP
  PRI
END
```

```
Make EXE File
EXE File Name: P2.EXE

[ ] Produce Debug Code          Produce:
                                (•) EXE Requiring BRUN45.EXE
                                ( ) Stand-Alone EXE File

< Make EXE >  < Make EXE and Exit >  < Cancel >  < Help >
```

```
F1=Help  Enter=Execute  Esc=Cancel  Tab=Next Field  Arrow=Next Item
```

2-3 Make EXE dialog box.

of these files, put the appropriate utility disk in a drive and give QuickBASIC the full path name. (For example, if you have BC.EXE in the root directory of a disk you put in the A drive, enter A:BC.EXE.) QuickBASIC will do the rest. However, while this process is going on don't be surprised if the output screen fills with some cryptic messages. (You'll learn more about what these messages mean later.) In any case, sending information to a disk takes much more time than QuickBASIC's usual "all in memory" compilation—there's no reason to do this until you're happy with the way the program runs and need a standalone version for some reason. (One good reason is that a standalone version will work even faster than the program does from within the QuickBASIC environment.)

By the way, if you do return to DOS and use the DIR command to look for the executable version of this program, you might be shocked at how much space this simple program takes up. My source code for this program is only 100 bytes (a byte is one character) but the standalone, executable version is 14752 bytes. The EXEcutable version of a QuickBASIC program is always many thousands of bytes larger than the original source code. The reason for this is that a standalone, compiled program contains a large amount of "overhead"—information that might not be needed for a particular program but nonetheless is important for QuickBASIC's proper functioning. (From another point of view, a standalone QuickBASIC program takes up less space than an interpreted program. This is because you can run it without wasting memory for the fairly large BASIC interpreter.)

Why not do this check? Assuming you have enough memory (640K to play it safe, 512K should also work) then as usual, QuickBASIC makes the common programming practice of momentarily returning to DOS easy. Temporarily turn on the Full menus and move to the full Files (Alt+F) menu, press D to activate the DOS (Operating System) shell. This option lets you leave QuickBASIC and drop down to DOS. Once there, you can run most common (small) programs, for example, the DIR command or the executable version of CH1\P1.BAS. The DIR command is the easiest way to check that the EXE version really is as long as I said it was.

When you finish doing what you want to do at the DOS level, enter EXIT from the DOS prompt. This should take you back to QuickBASIC. Programs that you run from this DOS shell have to be small enough to fit into available memory along with QuickBASIC itself—otherwise, you won't be able to return to QuickBASIC. Also if you are running DOS 2.0 or 2.1 then I can't recommend using the DOS shell command—it seems to work erratically for these early versions of DOS and sometimes your machine "locks up."

Calculations: The Immediate window

Here are the symbols QuickBASIC recognizes for the five fundamental arithmetic operations:

+	addition
−	subtraction
/	division
*	multiplication (don't use an x)
^	exponentiation

One way to do calculations is to incorporate these symbols into a PRINT statement and use the Immediate window. Use F6 to make this window active (as always the cursor marks the active window). Among other things, the Immediate window allows you to have your computer imitate a $10 calculator. For example, type:

```
PRINT 2 + 2
```

in the Immediate window, and when you hit Enter you'll see a "4" in the output window.

More precisely, the Immediate Window allows you to enter up to ten lines. Any one of them will be executed by using the arrow keys to place the cursor on the line, and hitting Enter. For example, you might want to enter a CLS in the Immediate window and hit Enter to clear the output screen. Because you can use any QuickBASIC commands in the Immediate window, another use of the Immediate window is to give a quick way to make a directory without leaving the

QuickBASIC environment. The command in QuickBASIC is also MKDIR, only now the name of the directory is enclosed by quotes. So:

```
MKDIR "C:\CH1"
```

would make a directory on the C drive called CH1. I'll have more to say about these types of commands in the file chapters.

If you want to do more complicated calculations, then I suggest using parentheses. Parentheses let you order mathematical operations. Something like 4+(3*5) gives 19 because first you do the operation within the parentheses (3 times 5) and only then add the 4. On the other hand, (4+3)*5 gives 35, because first I add the 4 and the 3 to get 7, and only then multiply by 5.

As an example:

```
((4 * 6) + 5) * 3
```

The innermost parentheses give 24, the second set tells you to add 5 to get 29 and finally multiply by 3 to get 87.

QuickBASIC allows you to avoid parentheses, providing you carefully follow rules that determine precedence. For example, multiplication always has precedence over addition. This means that 3+4*5 is 23 rather than 35. (Because multiplication has higher precedence than addition, QuickBASIC first does the multiplication 4*5 before doing the addition.)

Here is the order (hierarchy) of operations:

1. Exponentiation
2. Multiplication and Division
3. Addition and Subtraction

Think of these as levels. Operations on the same level are performed from left to right. So 36/6*2 is 12 (because division and multiplication are on the same level, first the division is done, giving 6, and then the multiplication). On the other hand, 36/3^2 is 4. This is because first the exponentiation (level one) is done, yielding 9 and only then is the division performed.

To show you how complicated using the hierarchy of operations can be, try to figure out what QuickBASIC will do with:

```
4*2+16/8+2^3^4
```

Here is what happens: First the exponentials (level 1) are computed left to right (2^3 = 8, 8^4 = 8*8*8*8 = 4096). Now, move to level 2 operations, and work left to right. Multiply 4*2 to get 8 and divide 16 by 8 to get 2. Finally add (lowest level!) them all together to get 4106.

All in all, I hope I've convinced you that a judicious use of parentheses will make your life easier.

Variables

QuickBASIC is quite flexible about variable names. They must begin with a letter but then can contain any number of letters, digits and periods (except it may not begin with the letters FN—see chapter 4). No other characters are allowed. 1BASE is an illegal variable name (because it begins with a 1) but BASE1 is all right. Because no special characters, save a period, are allowed, BASE#1 is forbidden but BASE.1 is acceptable. Also, case is irrelevant—Price, price and PRICE name the same variable but, as mentioned before, the Smart Editor mimics the last version you choose. Meaningful variable names are the best way to clarify the point of many program statements.

To swap the values of two variables, use (naturally enough) the SWAP command: enter `SWAP A,B` and QB sets up a temporary storage place to allow the swap.

An identifier at the end of the variable's name is used to indicate what kind of values that variable may have. The identifier for characters (strings) is a dollar sign ($). For example:

```
LET IBM$ = "Big Blue"
PC$ = "PS/2"
```

In QuickBASIC, a single string variable can hold up to 32,767 characters—far more than were allowed in earlier interpreted BASICs. If you store numbers inside a string variable, then what you have is considered only as a group of digits so PRINT "2" * "2" makes no sense (and will give you a compile time error "type mismatch").

Use:

% to denote an integer variable
& For a long integer variable
! Single precision
# Double precision

The only identifiers that may not be familiar to you are the ones for "long integers" and "double precision." These types are new to QB. The first is used for variables holding integer values between −32,768 and +32,767. The latter can hold as large as 308 digits—accuracy is still restricted to 15 or 16 digits, however.

One problem with the different kinds of variables is that you might, by mistake, leave off the identifier. One way to avoid this problem is to use QuickBASIC's ability to specify that all variables beginning with a single letter (or range of letters) will only be used for a single type of variable. This overrules QuickBASIC's default definition of variables as single precision.

For example, to make all variables beginning with the letter I, integer variables (so I, IZZY,ICHABOD would automatically be integer variables), place the command DEFINT I in your program before you use any variables beginning with I. Similarly, to make all variables whose names begin with I, J or K integer variables, use DEFINT I-K. DEFINT A-Z then makes all variables default to integer variables. In a sense, the DEFINT A-Z command changes QuickBASIC's built-in default options.

However, just as for QuickBASIC's built-in defaults, you can always overrule this command by using the correct identifier. Even after a DEFINT A-Z, A# is still a double-precision variable. Here are the other DEFiners

DEFLNG	(to define a long integer range)
DEFSNG	(to define a single precision range)
DEFDBL	(to define a double precision range)
DEFSTR	(to define a string range)

just as for the DEFINT command you can use the dash to define a range for variables of a fixed type: DEFDBL X-Z makes all variables beginning with X, Y or Z double-precision.

In general, most programmers use one of these DEFiners only if most of the variables in the program are of a single type. DEFINT A-Z, for example, would be used if a programmer was convinced that they would be using mostly integers.

If you've tried any calculation involving very large numbers, then you have probably found out that QuickBASIC often doesn't bother printing out large numbers of insignificant zeros. Instead, it uses a variant on scientific notation. For example, if you ask QuickBASIC to print a 1 followed by 30 zeros (using, say PRINT 10^30) then what you end up with is 1E+030. If you are not familiar with this notation, then think of the E+ as meaning "move the decimal place to the right and add zeros if necessary. But how many places should you move the decimal point? It's exactly the number following the + sign. When negative numbers follow the E, move the decimal point to the left: 1.7E−5 gives you .000017.

QuickBASIC starts using the E notation if the number requires more than 16 places to display. However, you can enter a number using the E notation any time you find it easier to do, so QuickBASIC doesn't care whether you enter 1000, 1E3 or 1E+3. (To make the number double-precision, use "D" instead of the "E.")

Finally, if you assign a single-precision variable to a double-precision variable, you do not suddenly increase its accuracy. The number may suddenly have more (or different) digits—but only the first six or seven can be trusted.

(Assigning a single-precision variable to a double-precision variable is an example of what programmers call *type conversion*. This means that QuickBASIC changes the type of a variable if it has to. If A% = 3 then the assignment statement A# = A% would involve changing the 3 to a double-precision value in order for this statement to make sense.)

When you use a constant in a program, QuickBASIC assumes that:

- If a number has no decimal point and is in the range −32768 to 32767, then it's an integer.
- If a number has no decimal point and is in the range for a long integer, then it's a long integer.
- If a number has a decimal point and is in the range for a single-precision number, then it's assumed to be single-precision.
- If a number has a decimal point and it's outside the range for a single-precision number, then it's assumed to be double-precision.

These assumptions occasionally lead to problems. This is because the realm in which an answer lives is determined by where the questions lived. If you start with two integers, QuickBASIC assumes the answer is an integer. For example, in a statement like

```
PRINT 23456*12345
```

QuickBASIC assumes both 23456 and 12345 are integers. The answer is also assumed to be an integer—but, of course, it's too big and this statement would generate an overflow error. The cure is to add the double-precision or long integer identifier to one of the numbers. Write:

```
PRINT 23456#*12345
```

This forces QuickBASIC to treat the 23456 as a double-precision number and the answer as well.

Another way to force a type conversion is to use special built-in functions. CINT would convert a small enough number to an integer by rounding. CLNG would convert it to a long integer by rounding. CSNG converts to single-precision and CDBL to double-precision.

One final point is worth keeping in mind: You cannot use the SWAP command to interchange the contents of two variables of different types. SWAP Integer%,A& is impossible—even if the value of A& happened to be within the proper range for an integer.

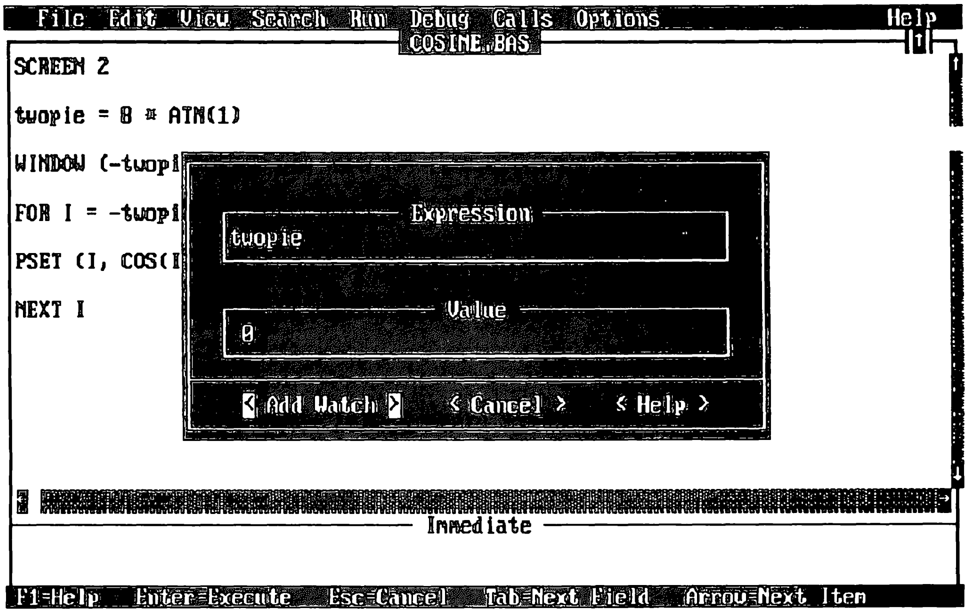
Debugging a program

Before I start, you may be interested in the story of the first computer bug. Early computers used vacuum tubes and these tubes generated a lot of heat. For this reason, the windows were often left open. When one of the earliest computers (the MARK II) was not working, a repairman found that a moth had wandered in and had gotten caught in a relay—which caused a short circuit. Grace Hooper (one of the most important of the early computer scientists) remarked that the removal of the moth was the "First successful debugging" and she taped the moth to the computer's log book.

Whenever you try to debug a program, you're best off having a hardcopy of the source code. Check each statement carefully. Are all the variables the ones you want? Are any spelled incorrectly (leading to default values that foul up the results)?

QuickBASIC has extraordinarily powerful debugging facilities that are available via the Debug menu. I want to begin with two that work well together. The first is called "setting a breakpoint." A breakpoint is a place where the program stops for a moment. Hitting F5 (the shortcut for the continue option on the Run menu) causes the program to resume.

Why is it useful to be able to temporarily stop a program? Well, imagine that you are concerned with why a variable seems not to have the value that you want at the end of a program. Set a breakpoint at some earlier place. Breakpoints are toggled via F9 or B on the debug menu. At this point, you can use the next feature I mentioned: Instant watch. This lets you examine the value of any variable of a halted program (or in certain situations, a program that has just ended). To do this, wait until the program stops when it hits the breakpoint. Breakpoints are shown in reverse video or highlighted. Now place the cursor on the variable whose value you want to examine. Activate this feature via its shortcut key—Shift+F9 (or I on the Debug menu). A window pops up that tells you the current value (see Fig. 2-4).


File Edit View Search Run Debug Calls Options Help
COSINE.BAS
Immediate
< Add Watch > < Cancel > < Help >
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

```
SCREEN 2
twopie = 8 * ATN(1)
WINDOW (-twopi
FOR I = -twopi
PSET (I, COS(I
NEXT I
```

Expression: twopie

Value: 0

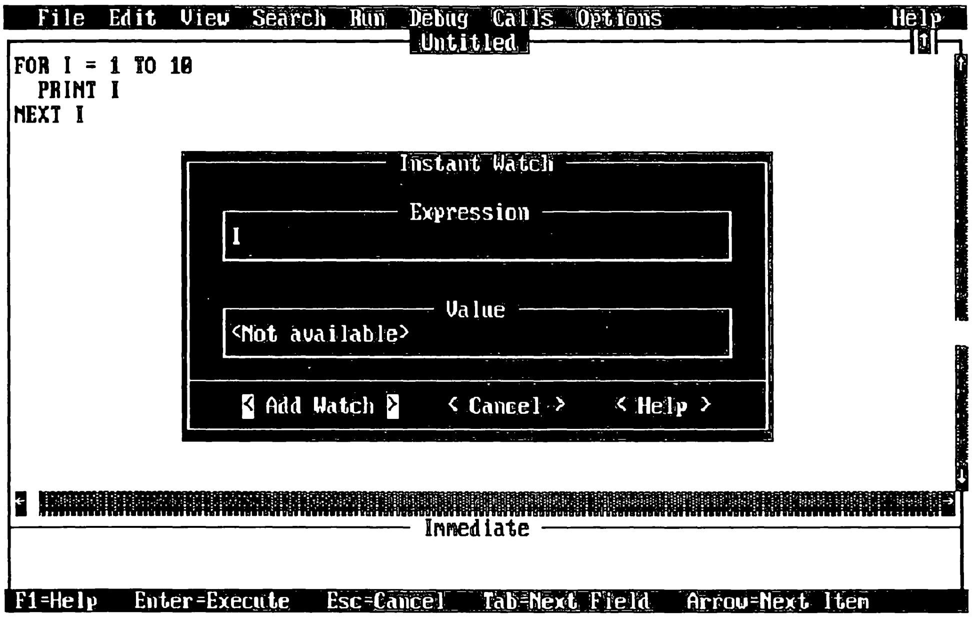
2-4 The instant watch dialog box.

The options are: Add Watch and Help. The Add Watch option allows you to monitor the value of a variable, because it may change—ideal for debugging a program with a FOR-NEXT loop.

As an example, look at the following simple program:

```
FOR I = 1 TO 10
    PRINT I
NEXT I
```

Enter this and, when you're done, position the cursor on the variable I and hit the instant watch shortcut (Shift+F9). The screen should look like Fig. 2-5.


File Edit View Search Run Debug Calls Options Help
Untitled
Instant Watch
Expression
Value
< Add Watch > < Cancel > < Help >
Immediate
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

```
FOR I = 1 TO 10
  PRINT I
NEXT I
```

I

<Not available>

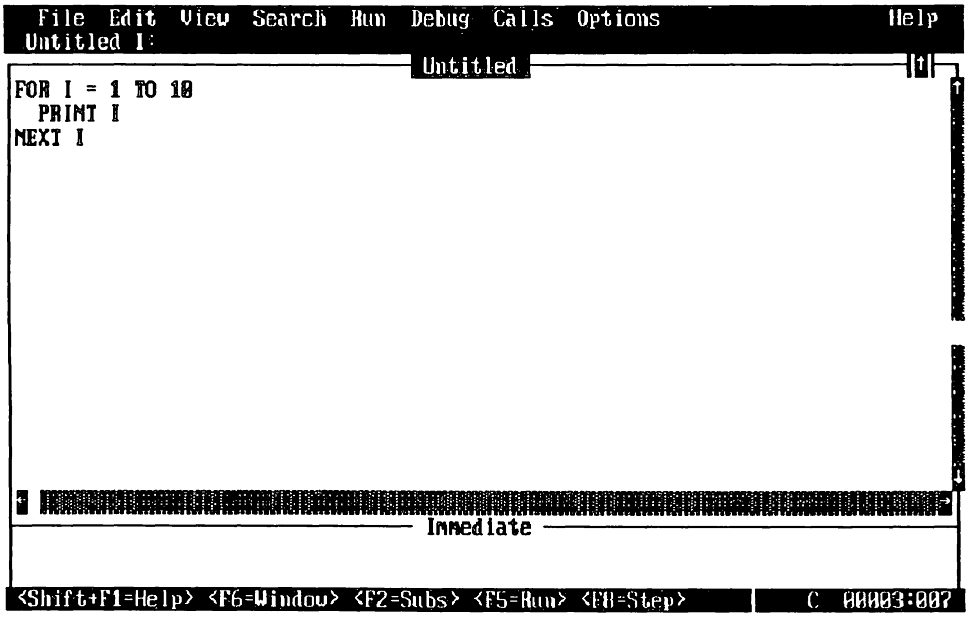
2-5 The instant watch screen.

Notice the cursor is at the Add watch option. Hit Enter to accept this option. The dialogue box closes and the screen looks like Fig. 2-6.

Notice the small window on the top has opened up and it says:

Untitled I:

This will give the current value of the variable I in this program. Now toggle a breakpoint at the PRINT I statement (F9). Finally, run the program. Of course, the breakpoint stops the program on each pass through the loop. Notice, however, that each time you use the continue (F5) key to make another pass through the loop, the value of I in the Watch window has changed. In fact, although I didn't mention it in the last chapter, once you've set up a breakpoint you can now single


File Edit View Search Run Debug Calls Options Help
Untitled I:
Untitled
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
C 00003:007

```
FOR I = 1 TO 10
  PRINT I
NEXT I
```

2-6 Screen with a watch window.

step through your program. This is done via the F8 key. Notice that each time you press the F8 key, the next statement is highlighted.

Because the ability to single step through a program is so important, QuickBASIC gives you a way of doing this without having to go through the trouble of setting up a breakpoint. This is done via the Run menu. Instead of using the Start option, use the Restart option. In spite of its name this does not require having already started the program. What it does is take your program to the starting gate: all variables are reset to their starting values, and QuickBASIC moves back to the top line (which is highlighted). Now you can use the F8 key to single step through the program. As you single step, each successive statement is highlighted. Combine this with the Add watch option available via Instant watch (Shift+F9) or on the Debug menu and you now have a method to watch how the value of variables evolve. By going back and forth to the output screen via F4 at each step, you also have a method of watching the output screen evolve.

Sometimes a judicious temporary insertion of a command new to QB 4.5 can help as well: this is the SLEEP command. In earlier versions of BASIC people were often forced to use "empty FOR-NEXT loops" or "while loops" for timing purposes. In QB 4.5 (as long as you want to wait an integral number of seconds, i.e., 1, 2, 3, 4 etc.), the SLEEP command can replace them. A statement like: SLEEP 5, stops a program in its tracks for 5 seconds; SLEEP .1 should stop it for a tenth of a second but, unfortunately, it doesn't. The SLEEP command rounds the value and always waits an integral number of seconds.

The SLEEP command has another use—the command SLEEP without any time specified suspends the program until you press a key. Keep in mind that a command like SLEEP 5 is also overridden by a key press. If you want the program to wait 5 seconds without risking the "sleep" being interrupted, then you'll need to write a timing loop.

QuickBASIC has a way to break (stop) most any loop. Hit the Ctrl+Break combination. Fortunately, unlike any earlier compiler, once you Break a program, there's usually still a way to continue. Just as with the breakpoints from chapter 3, it's F5 or continue on the Run menu. (For more on this useful technique see the last section of this chapter.) With a standalone program, on the other hand, there's no way to continue once you've Broken it. In fact, if you don't toggle on the "Produce debug code" (or make sure that it was toggled on) when you create the compiled version, then you can only stop an infinite loop by a system reset.

You should keep in mind that standalone programs with the keyboard+break option (Produce Debug Code) enabled (computerese for turned on) will run slightly slower than those compiled with it off. QuickBASIC needs to check constantly (poll) the keyboard to see if someone presses the Ctrl+Break combination. For that reason, after the debugging process is over and you're sure you won't have any infinite loops, then you might want to turn it off before the final compilation.

An example—controlling a printer

Modern printers are quite versatile: most can print tiny letters (17 or 20 characters to the inch) or enlarged letters (5 characters to the inch). They can usually print in italics and underline text as well. Many can even beep if you send the LPRINT CHR$(7) command. It's quite possible, in fact, to control where each dot is placed on a line and how wide the spaces between lines will be. All these powers are invoked by sending special codes to the printer. For example, you might want to have the printer automatically move to the top of the next page after it's finished printing a document. This is usually called forcing a form feed, and all it involves—on every printer that I've ever encountered—is sending a Ctrl+L (ASCII code 12) combination out. This is easily done by the statement:

```
LPRINT CHR$(12)
```

The codes that control the more sophisticated features vary from printer to printer. However, the most common codes are those belonging to the family of EPSON (IBM graphics) compatible printers. For example, on these printers one turns on the large (5 characters to the inch) printing by sending a Ctrl+O (CHR$(15)) combination to the printer. To turn it off, you send out a Ctrl+R (CHR$(18)). Here's a table of some of the more useful Epson compatible control codes.

BELL	CHR$(7)
BACKSPACE	CHR$(8)
LINE FEED	CHR$(10)
FORM FEED	CHR$(12)
CARRIAGE RETURN	CHR$(13)
COMPRESSED ON	CHR$(15)
COMPRESSED OFF	CHR$(18)
DOUBLE WIDTH (5 CPI)	CHR$(14)
DOUBLE WIDTH OFF	CHR$(18)

Note that modern printers distinguish between a line feed, a carriage return, and the combination of the two. A line feed moves the paper up one line but doesn't move the print head. A carriage return moves the print head to the left margin, but doesn't move the paper. The carriage return, line feed combination (CHR$(10) + CHR$(13)) is what some people mean when they talk informally of a "carriage return."

The compressed mode usually gives you 132 characters per page. It's useful for displaying wide tables (spreadsheets)—especially when you have a normal sized printer. The double width characters make nice headlines but be aware that, unlike the compressed characters, the CHR$(14) command that turns them on only works for the current line.

Another way to control a printer involves sending it more than one control code at a time. Most of these combination codes are preceded by the special code (ASCII 27) that the keyboard generates when the Esc key is pressed. For this reason they are usually called *escape sequences*. Here's a list of some of the more common escape sequences for Epson and Epson compatible printers.

ITALICS ON	CHR$(27)+CHR$(52)
ITALICS OFF	CHR$(27)+CHR$(53)
UNDERLINE ON	CHR$(27)+CHR$(45)+CHR$(1)
UNDERLINE OFF	CHR$(27)+CHR$(41)+CHR$(0)
DOUBLE STRIKE ON	CHR$(27)+CHR$(71)
DOUBLE STRIKE OFF	CHR$(27)+CHR$(72)
SUPERSCRIPT ON	CHR$(27)+CHR$(83)+CHR$(0)
SUBSCRIPT ON	CHR$(27)+CHR$(83)+CHR$(1)
SUBSCRIPT/SUPERSCRIPT OFF	CHR$(27)+CHR$(84)
PROPORTIONAL SPACES ON	CHR$(27)+CHR$(112)+CHR$(1)

PROPORTIONAL SPACE OFF	CHR$(27)+CHR$(112)+CHR$(0)
MASTER RESET (cancels all active features)	CHR$(27)+CHR$(64)

Older printers might not support proportional spacing. (Support being computer-ese for have available.) Proportional spacing means that letters like an m take up more space than letters like i. This gives a cleaner, more typeset look. Some printers let you combine these features—for example, printing in proportionally spaced italics or compressed superscripts. Obviously, printers are different and the manual that came with your printer is the best guide. I've only given you a sample of the features. You can control each pin separately and, most likely, have very fine control of the spacing between lines as well.

Here's a demonstration program that uses the control codes to print the same string in different ways.

```
' CH2\P2.BAS
' some common features of Epson compatible printers

CLS
PRINT "This demonstrates some common features of an Epson"
PRINT "compatible printer."
PRINT
PRINT "Enter a sentence - no longer than 80 characters"
INPUT A$

LPRINT CHR$(12)                     ' New sheet of paper

LPRINT CHR$(27);CHR$(52);A$         ' start italic
LPRINT CHR$(27);CHR$(53)            ' cancel italic

LPRINT CHR$(27);CHR$(71);A$         ' start double-strike
LPRINT CHR$(27);CHR$(72)            ' end double-strike


LPRINT CHR$(27);CHR$(45);CHR$(1);A$  ' start underlining
LPRINT CHR$(27);CHR$(45);CHR$(2)     ' end underlining

LPRINT CHR$(27);CHR$(112);CHR$(1);A$ ' proportional space on
LPRINT CHR$(27);CHR$(112);CHR$(0)    ' proportional space off

LPRINT "Water!":LPRINT
LPRINT "H";CHR$(27);CHR$(83);CHR$(1);"2";  'subscript demo
LPRINT CHR$(27);CHR$(84);                  ' subscript off
LPRINT "O"
LPRINT

LPRINT "And finally - heavy water!":LPRINT
LPRINT "H";CHR$(27);CHR$(83);CHR$(1);"2";
LPRINT  CHR$(8);                          'back space
LPRINT CHR$(27);CHR$(83);CHR$(0);"2";     'superscript
LPRINT CHR$(27);CHR$(84);
LPRINT "O"

LPRINT CHR$(27);CHR$(64)              ' back to default

END
```

I suspect that this program is pretty clear. I simply send the appropriate control code out before printing the phrase (except maybe printing the symbol for heavy water, H^2_2O. Heavy water, or deuterium oxide, is water where the ordinary hydrogen is replaced by deuterium. The symbol for deuterium is H^2. To get a superscript and subscript in (essentially) the same place, I use the backspace code (CHR$(8)) to move the print head back one space.

On the other hand, even if this program is pretty clear, it's also written pretty stupidly. I've broken a cardinal rule: Write programs so they are as easy as possible to correct. Here the lines are made up of multiple CHR$ commands and it's far too easy to make a mistake. A far better written program would begin by assigning variables to the printer features I wanted to control:

```
FormFeed$ = CHR$(12)                   ' New sheet of paper
ItalOn$ = CHR$(27)+CHR$(52)            ' italic
ItalOff$ =CHR$(27)+CHR$(53)            ' cancel italic

DoublStr$ = CHR$(27)+CHR$(71)          ' double-strike
DoubleOff$ =CHR$(27)+CHR$(72)          ' end double-strike
```

Now I would use these variables; instead of saying:

```
    LPRINT CHR$(27);CHR$(52);A$     ' start italic
```

I would say:

```
    PRINT ItalOn$;A$
```

and so on.

Besides making the program easier to read and correct, using variables instead of codes can often make it easier to adapt a program to a new situation. In this case, to use the correctly written program for other printers, all I have to do is change a few assignment statements—which are prominently displayed at the beginning of the program. I won't need to search through many lines of text and make multiple changes. QuickBASIC allows you to set up "named constants" for situations like this—more on them later.

Pretty printing

You've already seen a problem with calculations done by QuickBASIC: the answers to simple calculations look strange. You end up with 16 decimal digits when you really want an answer to look like .37! You overcome this problem with a variant on the PRINT (LPRINT) command called PRINT USING (LPRINT USING).

This command works with templates (they're also called *format strings*). For numbers, these templates usually consist of number signs (#) with a decimal point followed by more # signs. For example, the command:

```
PRINT USING "###.##";438.37123456
```

yields

```
438.37.
```

Moreover, the compiler will, if necessary, round the number so that it fits into the template. If you ask it to:

```
PRINT USING "###.##";438.37649678
```

what you will see is:

```
438.38
```

because this number is rounded up (because the digit after the 7 is a 6) to fit into a template having only two decimal places.

Similarly, if you use the PRINT USING command, and follow it by a calculation, then the computer will do any rounding necessary to fit the result inside the template. For example, consider the following program fragment:

```
CLS

PRINT 1000/7
PRINT USING "###.##";1000/7
```

What you will see is:

```
 142.8571

142.86
```

Notice that the display is indented one space for the first calculation and flush left for the answer displayed using the PRINT USING command. This is because the PRINT USING command, unlike the PRINT command, does not leave room for an implied + sign. If you want a plus or minus sign in front of a number, then you must put it in front of the format string. For example,

```
PRINT USING "+###.##";342.71
```

yields:

```
+342.71
```

The result of using a format string can be complicated—we'll work through the following example without using the computer. Suppose your template has five #'s before the decimal place and three #'s after. Then the PRINT USING "*#####.###*" command tries to print any number using no more than five digits before the decimal point and three digits after. It will try to print any number in a nine space block. If there are fewer than three decimal digits after the decimal point, then trailing zeros will be printed. If there are fewer than five digits in front of the decimal point, then leading blanks will be printed. Thus, printing more than one number with the same template always lines up the decimal points—extra spaces are placed to the right and left as needed.

What happens if you try to print a number with more than five digits in front of the decimal point? Say

```
PRINT USING "#####.###";123456.123?
```

QuickBASIC prints the entire number, but precedes it with a % to tell you that the format string didn't fit the number.

The leading blanks that the PRINT USING command leaves can be a problem. Suppose you were writing a program that will print out checks. You obviously don't want to have any leading blanks; they are an invitation for someone to fill in a larger amount. Another kind of format string takes care of this. If you place two dollar signs ($$) in front of any format string, then a dollar sign is placed flush with the beginning of the number—no leading spaces will show up. One of the two dollar signs stands for the dollar sign, the other plays a dual role. The statement PRINT USING "###.##" allows numbers as large as 999.99; however the statement PRINT USING "$$###.##" allows (dollar) amounts as large as 9999.99.

Similarly, if you place two asterisks (**) in front of a format string, all leading spaces are replaced by asterisks. Here, however, both asterisks count as digit positions so:

```
PRINT USING "**###.##"
```

allows a number as large as 99,999.99 (although, in printing a number as large as this, no asterisks would be used).

Finally, if you want to put commas after every three digits within your numbers, use a comma before the decimal point. For example:

```
PRINT USING "$$###,.##";9999.89
```

gives:

```
$9,999.89
```

Scientists sometimes like numbers to be printed in exponential notation. They like 1E03 better than 1000. If you need to do this, place four (or five if you have too many digits in the exponent) exponential signs (^) after the template: PRINT USING "#.##^^^^";1000 gives 1.00E03.

You can place a group of characters in the quotes along with the template, and they will be reproduced exactly. For example:

```
PRINT USING "I will pay you $#######,.## if ... "; 1000000
```

gives

```
I will pay you $1,000,000.00 if ...
```

Here's an example of where this would be needed: Suppose you wanted to write a program that would convert numbers to percents. At the key conversion step you can't say:

```
PRINT Number*100;"%"
```

because the automatic space that QuickBASIC sticks on at the end of the calculation would force the % sign over by a space. Instead, assuming you want only two decimal points, say:

```
PRINT USING "###.##%";Number*100
```

Another way to put a string at the end of a number is to follow the PRINT USING statement by a semi-colon. This accomplishes the same thing as the previous example, because the semi-colon suppresses the automatic carriage return here as well.

Finally, let me end this section by pointing out that programs that use complicated format strings are as badly written as the original printer control program, and for much the same reasons. The cure is also the same. Use string variables for your format strings. The STRING$ command combined with the + (concatenation operator) makes setting up these variables easy. Instead of saying:

```
PRINT USING "$$#######,.##"
```

use an extra statement:

```
LargeNumber$ = "$$" + STRING$ (7,"#") + ",." + STRING$(2,"#")
PRINT USING LargeNumber$;
```

The PRINT USING command is so important, and the number of options and possibilities so large, that I find myself frequently referring to the on-line help for this command. Luckily, it includes a complete, detailed summary of its many variations.

IF-THEN ELSE and the BLOCK IF

Suppose you want to write a program that accepts two numbers and then prints them out in the correct (numerical) order. Your first instinct might be to write something like:

```
INPUT A,B
    IF A > = B THEN PRINT B,A,
    PRINT A,B
```

This seems to work. If A is less than B, then the test is unsuccessful and the THEN clause (the PRINT statement) is disregarded. We print first A then B as we're supposed to. The problem occurs because the second line: PRINT A,B is always processed. In this situation you really want to mimic the English construction: "IF A is greater than B, then print B followed by A ELSE (otherwise) print A then B." This important variant of the IF-THEN exists in QuickBASIC and is called the IF-THEN ELSE. The fragment should read:

```
INPUT A,B
IF A > = B THEN PRINT B,A ELSE PRINT A,B
```

When QuickBASIC processes an IF-THEN ELSE, it does the test, and if the test succeeds it processes the command that follows the key word THEN (the THEN clause). If the test fails, it processes the statement that follows the key word ELSE (naturally enough: the ELSE clause).

When you start using the IF-THEN ELSE you quickly run past the limits of an 80 column line. There are various ways around it. I feel that the worst choice is to use the 255 character limit on a logical line in QuickBASIC and just keep on typing. This may take advantage of QuickBASIC's neat (but superfluous?) horizontal scrolling ability, but you give up the ability to see what's going on. (Your printouts will look horrible as well.)

The best choice is to use the most powerful form of the IF-THEN command that's available in QB. It's called the block IF-THEN. This lets you process as many statements as you like in response to a true condition:

```
If I win the lottery THEN
    I'M happy
    My family is happy
    And, the tax man is happy.
```

Here I have three statements in response to something being true. The way QuickBASIC translates this construction, the block IF-THEN-ELSE has a slightly different format than the usual IF-THEN. It looks like:

```
IF thing to test THEN
    lots of statements
ELSE
    more statements
END IF
```

Now you do not put anything on the line following the key word THEN—hit the Enter key immediately after typing it. This bare "THEN" is how QuickBASIC knows it's beginning a block. The word ELSE is optional. Putting it there (again, alone on a line) means that another block will follow—to be processed only if the IF clause is false. However, whether the ELSE is there or not, the "block IF" must end with the key words END IF.

For an example of this, suppose you work for a company and the bonus you get for selling widgets (and the discount the customer gets) depends on the number of widgets they buy. If you sell at least 1000 widgets, then they get a 40% discount and you get a $500 bonus. Otherwise they get a 35% discount and you get a $100 bonus. Here's a fragment for this:

```
INPUT "How many widgets did you sell";WidgetCount

IF WidgetCount > =1000 THEN
     Discount = .4
     BONUS = 500
ELSE
     Discount = .35
     BONUS = 100
END IF
```

Again, as usual, the indentation is there to make the program more readable—QuickBASIC doesn't care.

SELECT CASE

Programs often deal with many possibilities—with many branches. When this happens, you can always use multiple IF-THEN's to eliminate all but one of the possibilities. However, most programmers feel that the IF-THEN ELSE should be reserved for a program fork—a place where you have two choices, not 14.

When you have multiple branches, QuickBASIC has the SELECT command. To use this command, you start with something you want to test. For example, suppose you want to test if a character is a vowel. You could write:

```
IF UCASE$(CHAR$) = "A" THEN PRINT "Vowel"
IF UCASE$(CHAR$) = "I" THEN PRINT "Vowel"
```

etc.

Using this new command you write this:

```
SELECT CASE UCASE$(CHAR$)

  CASE  "A"
   PRINT "Vowel"
```

```
CASE  "E"
 PRINT "Vowel"

CASE  "I"
 PRINT "Vowel"

CASE  "O"
 PRINT "Vowel"

CASE  "U"
 PRINT "Vowel"

CASE  "Y"
  PRINT "Y is a problem - sorry"

END SELECT
```

The SELECT CASE command makes it clear that a program has reached a point with many branches—multiple IF-THEN's do not (and the clearer a program is, the easier it is to debug). QuickBASIC does at most one CASE—the first one it finds to be true.

What follows the key word SELECT case is a variable or expression—and what QuickBASIC is going to do depends on the value of the variable or expression. The key word CASE is shorthand for "In the case that the variable (expression) is" and you usually follow it by a relational operator. For example, to begin to check that the value of the variable CHAR$ is a letter, you can add the case:

```
CASE IS < "A"
PRINT "Character is not a letter."
PRINT "Meaningless question"
```

To eliminate all possible non-letter values you have to screen out those characters whose ASCII codes are between 91 and 95. This is done as follows:

```
CASE CHR$(91) TO CHR$(95)
    PRINT "Character is not a letter."
    PRINT "Meaningless question"
```

Here the key word TO allows you to give a range of values. So this statement is shorthand for: In the case that the variable is in the range from CHR$(91) TO CHR$(95) inclusive. Finally, having eliminated the case when the character was not a letter, you want to print out the message that it is a consonant. This is done with the CASE ELSE (which is shorthand for: do this case if one of the other situations hold). Here's a more polished version of the program that incorporates all these conditions:

```
'CH2\P3.BAS
' A vowel tester revisited
```

```
CLS
PRINT "Press any key and I'll tell you if it's a consonant"
Char$ = INPUT$(1)
Char$ = UCASE$(Char$)                    'A

  SELECT CASE (Char$)

    CASE "A", "E", "I", "O", "U"          'B
     PRINT "Vowel"

    CASE "Y"
     PRINT "Y is a problem - sorry"

    CASE IS < "A"
     PRINT "Character is not a letter."
     PRINT "Meaningless question."

    CASE IS > "Z"
     PRINT "Character is not a letter."
     PRINT "Meaningless question."

    CASE ELSE                               'C
     PRINT "Consonant"

    END SELECT

END
```

A) I feel more comfortable doing the conversion outside the SELECT CASE statements. This makes it easier to make any changes needed to make this program part of some larger program.
B) QuickBASIC allows you to combine many tests for a single case. Just separate the tests by a comma. Think of the comma as "OR." This line replaces 10 or so lines of the original version; and, I think, makes the program more readable.
C) Here's the ELSE case—done in any remaining situations.

Finally, let me end this section with a program that lets you move the cursor around—much like the QuickBASIC editor does. This kind of program is frequently needed. Because you haven't yet seen how to read the arrow keys, I'll just mimic how the editor reads the control keys.

An outline is:

```
Initialize
    Make the cursor large
    move it to the top left hand corner.

Read a key-stroke

    Ctrl-D (ASCII 4)     move right 1 column
    Ctrl-S (ASCII 19)    move left 1 column
```

Ctrl-E (ASCII 5)	move up 1 row
Ctrl-X (ASCII 24)	move down 1 row
Ctrl-R (ASCII 18)	page up (In our program top row, same column)
Ctrl-C (ASCII 3)	page down (In our program bottom row, same column)

Stop when an X is hit.

Whenever your outlines have this form—lots of different CASEs—then you'll probably use the SELECT CASE statement in the translation.

Here's the program:

```
'CH2\P4.BAS
' A cursor mover

CLS
PRINT "This demonstrates the WordStar key movements.  Cntrl +";
PRINT " S, D, E, R and C and X "

PRINT "will move a large cursor around."
PRINT
PRINT "Press any key to start.  A capitol `X' ends."

KeyStroke$ = INKEY$
IF KeyStroke$ <> "X" THEN
 CLS
 KEY OFF
 LOCATE 1, 1, 1, 0, 7                  ' make the cursor large
 Row = 1:Col = 1
END IF

DO UNTIL KeyStroke$ = "X"

    DO                                    'A
     KeyStroke$ = INKEY$
    LOOP WHILE KeyStroke$ = ""

      SELECT CASE KeyStroke$

        CASE CHR$(4)                        'CNT/D   (B)
          IF Col <= 79 THEN Col = Col + 1
        CASE CHR$(19)                 'CNT/S
          IF Col > 1 THEN Col = Col - 1
        CASE CHR$(5)                       'CNT/E
          IF Row > 1 THEN Row = Row - 1
        CASE CHR$(24)                       'CNT/X
          IF Row < 25 THEN Row = Row + 1
        CASE CHR$(18)                       'CNT/R
          Row = 1
        CASE CHR$(3)                        'CNT/C
      Row = 25
     CASE ELSE                                              '(C)
      IF KeyStroke$ <> "X" THEN BEEP

     END SELECT

   LOCATE Row, Col

 LOOP

END
```

A) Waits until a key is pressed.
B) This is the first of the many CASEs needed to implement the outline. Obviously there are rather many of them but they should, I hope, all be straightforward.
C) I could have eliminated the small DO loop at the top by writing:

   ```
   IF KeyPress$ <> " " AND KeyPress$ < > "X" THEN BEEP
   ```

   but I'm sure you will agree that this negative test is confusing—so confusing, in fact, that when this kind of "double negative" test does come up (and it does!) I sometimes prefer to waste a statement and write something like:

   ```
   IF KeyPress$ = " " OR KeyPress$ = "X" THEN
     ' DO NOTHING
   ELSE
     BEEP
   END IF
   ```

   I personally find this much clearer. If the IF clause is true, then I want to do nothing (as the REMark statement indicates) and the ELSE clause is the translation of the complicated IF KeyPress$< > " " AND . . . . statement
D) The program ends when a capital X is pressed. In chapter 8 you'll see how to read the arrow keys. You might want to add the appropriate LOCATE command before the END command to reset the cursor back to normal. Or, use the Immediate window to do so.

Odds and ends of the IF-THEN

The SELECT case can only test one expression—ultimately one number or string. Suppose you have two numbers A, B and your outline looks like this:

```
IF A=B  DO . . . . .
IF A>B  DO . . . . .
IF A<B  DO . . . .
```

One way to program this is to set up a variable:

```
Difference = A - B
```

and then SELECT on whether the value of Difference was zero (when A = B), greater than zero (in which case A > B), or less than zero (A < B). This is probably the way I would do it. But now suppose someone throws in one or two extra conditions:

```
IF A > B AND A<2*B
IF A > 2*B
```

Now it's no longer obvious how to use the SELECT CASE command. You could write four block IF-THEN's corresponding to each of the different conditions in the outline and most of the time this wouldn't cause any problem. Problems may happen if you, in the example above, have to do something to "A" or "B" in one of the blocks. From that point on you're in trouble. All further tests are off. More precisely suppose the outline was:

```
Do one of the following:
IF A = B
    PRINT A
IF A < B
    PRINT A and add two to A
IF A > B
    PRINT B and add two to B
```

Here is a translation of this outline:

```
' CH2\P4.BAS
' A demo

INPUT "Two numbers please"; A, B

IF A = B THEN PRINT A

IF A < B THEN
  PRINT A
  A = A + 2
END IF

IF A > B THEN
 PRINT B
 B = B + 2
END IF

END
```

Suppose the value of A was four and the value of B was five. Then the second option is taken and the program prints out four and makes the value of A equal to

6. But now the third option is activated—contrary to the outline which says do only one of the possibilities.

This situation is similar to when we introduced the ELSE statement. You need to continue testing within the confines of the original IF-THEN. This is done with the command:

    ELSEIF . . . THEN

Here's the correct translation of the above:

```
IF A = B THEN
    PRINT A
ELSEIF A < B THEN
    PRINT A
    A = A + 2
ELSEIF A > B THEN
    PRINT B
    B = B + 2
END IF
```

Now everything is tied together. And, just like in the IF-THEN ELSE or the SELECT command, QuickBASIC activates at most one clause. In particular, if A<B then QuickBASIC processes only the second clause. When QuickBASIC is through doing that, it bypasses any other ELSEIF's that might be contained in the block—it goes immediately to the statement following the END IF. In this program it's the END command that finishes the program. By the way, you could replace the final ELSEIF by a simple ELSE—we've eliminated all the other possibilities.

A block IF-THEN can have as many ELSEIFs as you like but only one ELSE as the last clause. The limits are determined by how much you can process rather then what QuickBASIC can do. That's why, if I can, I prefer to use the SELECT CASE. Although any SELECT CASE can be transformed into an IF-THEN-ELSEIF, I find the latter much harder to read and hence to debug.

My final point is that the block IF-THEN is extremely flexible. You can put any QuickBASIC statement following the key word THEN—in particular, another IF-THEN ELSE. Consider the following:

```
IF Grade4 < 65 THEN
    PRINT "You failed the final exam."
    IF Average > 70 THEN
        PRINT "You pass because your average is sufficiently"
        PRINT "high to blot out failing the final exam."
    ELSE
        PRINT "I'm sorry failing the final and a marginal ";
        PRINT " passing average means failing the course"
    END IF
END IF
```

Is it clear that the ELSE belongs to the inner IF-THEN? The way to see this is to "play computer." For the ELSE to belong to the outer IF-THEN, the inner IF-THEN must have already finished. But it hasn't because, to that point, no END IF has shown up. So the first END IF finishes the inner IF-THEN and the second finishes the outer one and the ELSE must belong to the inner IF-THEN.

Loops

FOR-NEXT loops work much the same as in interpreted BASIC. The only difference is that you have a new way to get out of them: the EXIT-FOR. For example, suppose you want a FOR-NEXT loop to end abruptly—say when someone presses the Ctrl+C combination (ASCII code 3). This would take the form:

```
FOR counter = starting value TO test value
    IF INKEY$ = CHR$(3) THEN EXIT FOR
    .
    .
    .
NEXT counter
```

Recall the INKEY$ accepts a single character from the keyboard.

QuickBASIC has vastly improved methods of programming loops whose duration can't be determined beforehand. For example, suppose you wanted to start a program with a form of password protection. You want to prevent anyone from using a program until they enter the correct password. An outline for this kind of fragment might be:

```
ENTER PASSWORD
IF PASSWORD isn't right reenter PASSWORD
IF PASSWORD isn't right reenter PASSWORD
and so on.
```

This outline describes a repetitive operation—a loop—of a special sort: its length is unknown (and, in fact, after you start using these types of loops you must be especially careful that you haven't, inadvertently, set up an infinite loop). The commands in QuickBASIC needed for this kind of loop are quite close to English. In fact, assuming the password was my name, a fragment to do this is:

```
DO
    INPUT "Password please";Password$
LOOP UNTIL Password$ = "Gary Cornell"
```

The command DO starts a loop. QuickBASIC works its way through the statements that follow the keyword DO until it reaches the key words LOOP UNTIL. These key words are followed by something QuickBASIC can test. In this example, the test is if the password entered via the INPUT statement is indeed my name. If it is, then QuickBASIC ends the loop and moves on. If not, it goes back to the first statement following the keyword DO and repeats the process. Just like the FOR-NEXT loop, the statements that repeat are called the body of this loop. (It's usually called a DO UNTIL LOOP or simply a DO LOOP.)

Often you don't want the password to appear on the screen—to do this, change the INPUT to an INPUT$(12) and put the prompt inside a PRINT statement.

Compiling a program containing a fragment like this to a standalone (EXE) program without allowing the Ctrl+Break (turn the Produce Debug code option off) gives you a reasonably secure form of password protection. However, it's only reasonably secure because an expert programmer could break the protection—find the password—by carefully examining the EXE file. You'll see many more secure methods in later chapters.

It's important to remember that the test for equality is strict: entering "GARY CORNELL" would not work nor would "Gary cornell". Another thing to keep in mind is that the check is only done at the end of the loop. For example, if you change the fragment to read:

```
DO
    Password$ = "Gary Cornell"
    INPUT "Password please";Password$
LOOP UNTIL Password$ = "Gary Cornell"
```

Then, whether this loop ends still depends on the response to the INPUT statement. Initializing the variable Password$ to the correct value is irrelevant.

Obviously, something within the body of a DO LOOP that bears on the test had better change—otherwise, the test will always fail and you will be stuck in an infinite loop.

For more sophisticated programs you need ways to check for something besides equality. This is also done with the relational operators you use for IF-THEN's (<, >, <=, etc.). For strings, these operators test for ASCII order. This means that A comes before B, but B comes before a (and a space comes before any typewriter character). The string "aBCD" comes after the string "CDE" because uppercase letters come before lowercase letters.

Now suppose you want to add up the number of different types of insects that occur in North America. You take out your reference book and you observe that this list has to end with the "zyzzyva" (which my American Heritage dictionary defines as: "Any of various tropical American weevils of the genus Zyzzyva, often destructive of plants"). This makes your job easy:

```
' CH2\P5.BAS

CLS
InsectCount = 0

 DO
   INPUT "The next insect name";InsectName$
   InsectCount = InsectCount + 1
 LOOP UNTIL InsectName$ = "zyzzyva"

PRINT "The number of different types of insects is";InsectCount
END
```

This fragment seems like a prototype for many kinds of programs—ones that read in a list of names until the last one is encountered, keeping count all the while. On the other hand, it's a bit of a coincidence that the very last word in my dictionary is an American insect. Suppose that you wanted to count the number of names on a long list. To use a loop to keep count you need an ending name to test for. Usually you won't know the last entry so you're likely to use a group of strange characters (like "ZZZ") to act as a flag. Instead of testing for "zyzzyva" you test for a flag.

It's easy to modify the "InsectCount" program to test for a flag. This program does this:

```
' CH2/P5.BAS
' trying to count the names in a list

CLS
NameCount = 0

DO
  INPUT "Name - enter ZZZ when done";Entry$
  NameCount = NameCount + 1
LOOP UNTIL Entry$="ZZZ"

PRINT "The total number of names is";NameCount

END
```

The problem with this fragment is that it suffers from an off-by-one error. Imagine that the list consists of only one name beside the flag—what happens? Work through this program by hand: You enter the first name and the count increases to one. Next, you enter "ZZZ." However, because the test is only done at the end of the loop, the count increases to two before the test is done. So when the loop ends, the count is two when it ought to be one. One possible cure is to subtract one from the count once the loop ends. The trouble with this type of ad hoc solution (in computerese, a *kludge*) is that you end up having to constantly figure out how far off the results of your loops are when they finish—in order to move backwards.

Moving backwards is a bit silly when QuickBASIC makes the cure for this so easy: move the test to the top. Consider this listing:

```
'CH2/P6.BAS
' A counting program revisited

CLS
NameCount = 0
INPUT "Name - ZZZ to end";Entry$

DO UNTIL Entry$ = "ZZZ"
   NameCount = NameCount + 1
   INPUT "Name - ZZZ to end";Entry$
LOOP

PRINT "The total number of names is";NameCount

END
```

Now the user enters the first name before the loop starts. Once this is done, we do an initial test. Only if this test fails do we enter the loop and start adding one to the counter. (Notice that this kind of loop also works if there is nothing on the list—except the flag.)

A good rule of thumb is that, if you are going to use the flag, put the test at the end, if not put it at the beginning. More precisely: with the test at the end, the loop is always executed at least once, with it at the beginning, it may not be executed at all. Also remember that when the test is at the top, you obviously have to have something to test. Therefore: when the test is done at the beginning, initialize all variables before the loop starts. Finally, don't forget that you usually have to have two INPUT statements when the test is at the top: the first before the test and the second (to keep the process going) inside the loop.

You can use the second (test at the beginning) kind of DO LOOP to replace the FOR-NEXT loop. Doing this is a good way to test your mastery of the DO LOOP.

QuickBASIC has other kinds of loops, these loops consist of replacing the key word UNTIL with the key word WHILE. This new loop may seem superfluous: you can always change a DO UNTIL into a DO WHILE by reversing the relational operator:

```
DO
    UNTIL INKEY$ <> " "
```

is the same as:

```
DO
    WHILE INKEY$ = " "
```

or

```
    DO UNTIL Number > 5
```

is the same as:

```
    DO WHILE Number < = 5.
```

Given this why bother learning it? There are two reasons why I don't think the WHILE command is superfluous: the first is that, as much as possible, I want to write a program conforming to the way my mind works. And, sometimes I think of an operation as going on UNTIL something happens, while other times I think of it as continuing WHILE, as the saying goes, "the status is quo." The richness of QuickBASIC makes the fit between my internal thought patterns and the computer program I'm trying to write better.

It's not only me: psychologists have found that tests with positive conditions are easier to understand. DO UNTIL Number = 0 is easier to process than its counterpart: DO WHILE Number < > 0. Similarly, DO WHILE INKEY$ = " " is easier to process than DO UNTIL INKEY$ < > " ".

However, this is only a minor point in favor of the four possibilities—not enough to tip the balance. The best reason comes from when you have to combine conditions. This is usually done with the key words OR and the AND. These two key words work just like in English. We can continue a process as long as both conditions are true, or stop it when one turns false. For example, suppose we want to continue a program so long as the user presses a "Y" or a "y" at a key point. We can use the key word OR in the loop:

```
    DO
        Continue$ = INKEY$
    LOOP WHILE Continue$ = "Y" OR Continue$ = "y".
```

Now it's possible to translate this test so it works in an UNTIL LOOP—but it's certainly un-natural. (The translation is: LOOP UNTIL Continue$ < > "Y" AND Continue$ < > "y")

Let me end this section by warning you about a frequently occurring problem with these new kinds of loops. Consider the following program:

```
' CH2\P7.BAS
' a warning

Total = 0
PassNumber = 0

DO

 TOTAL = TOTAL + .1
```

```
PassNumber = PassNumber + 1
PRINT PassNumber,Total

LOOP UNTIL TOTAL = 1

END
```

You might think this program will end after 10 passes through the loop. It doesn't. In fact this fragment gives you an infinite loop and you'll need the Ctrl+Break combination to stop it. The reason we fell into an infinite loop is subtle but important. In this program all our numbers are values of single precision variables. And, as you now know, these numbers are only approximations. QuickBASIC's internal definition of .1 is off by a little in say, the seventh place. As you add .1 to the total, tiny errors accumulate and the resulting total gets awfully close to 1, but never exactly equals 1. Moral: Never test single or double precision variables for equality. This program should be rewritten to allow for tiny error:

```
' CH2\P8.BAS
' a warning
'revisited and corrected

Total = 0
PassNumber = 0

DO

 TOTAL = TOTAL + .1
 PassNumber = PassNumber + 1
 PRINT PassNumber,Total

LOOP UNTIL TOTAL > .99999999

END
```

Now the program really will stop after 10 passes through the loop. Among the various types of numbers you'll use in QuickBASIC programs, only test integers and long integers for equality. Single and double precision variables can only be checked to see if they are close (in computerese, to be within a certain tolerance).

Finally, to preserve compatibility with interpreted BASIC, QuickBASIC allows a variant on the DO WHILE LOOP (i.e., the test at the top). Instead of saying:

```
DO WHILE X = 0
LOOP
```

you can say:

```
WHILE X = 0
WEND.
```

Eureka loops—combining the IF-THEN with the DO-LOOP

Suppose we wanted to program a reading test so that it would clear the screen if either the "time was up," or if the person taking it indicated that they were finished (say, by striking a key). All we need to do is eliminate the SLEEP command and replace it with:

```
StartTime = TIMER: ENDTIME = TIMER
    DO UNTIL (EndTime - StartTime)> =DelyTime OR KeyPress < > " "
        EndTime = TIMER
        KeyPress$ = INKEY$
    LOOP
```

This DO-LOOP ends if either of two things happen: too much time has elapsed (much like the SLEEP command with a time interval would do), or someone presses a key (as the SLEEP command always does). You are probably wondering why bother with this construct: a SLEEP statement would do it faster. But suppose we needed to know exactly what happened—exactly why the loop ended. This is a situation tailor made for an IF-THEN. Merely add the following lines after the loop ended:

```
IF KeyPress$ < > " " THEN PRINT "Guessed too high!"
IF (EndTime-StartTime) > ~ DelyTime THEN PRINT "Guessed too low!"
```

Remember, you can't test a single precision variable for equality.

These kind of loops are so common they have a special name: eureka loops, after Archimedes famous bathtub experience. You set up a loop to end if either one of two things happens. Then you follow it by testing for which one actually happened.

Another way to combine the IF-THEN with a loop gives what I used to regard as the holy grail of loop writing: a clean, clear way to write a loop that tests in the middle. To do this, you combine the IF-THEN with a new command: the EXIT DO. For example, consider this re-written version of part of a grade book program:

```
DO

  INPUT "First exam grade - negative number to end";Grade1
  IF Grade1 < 0 THEN EXIT DO

  INPUT "Second exam grade";Grade2
  INPUT "Third exam grade";Grade3
  INPUT "Fourth exam grade";Grade4

  IF Average >=90 THEN PRINT "Grade is A"
  IF Average >=80 AND Average < 90 Then PRINT "Grade is B."
  IF Average >=70 AND Average < 80 Then PRINT "Grade is C."
  IF Average >=60 AND Average < 70 Then PRINT "Grade is D."
  IF Average < 60 Then PRINT "Sorry you fail.

LOOP

END
```

Notice that in this fragment there is neither a test at the top nor the bottom. The only test occurs in the form of an IF-THEN after the first grade is entered. It checks if the first grade is less than zero. If it is, then as usual, QB processes the THEN clause—in this case it's the statement EXIT DO. What this command does is pop you out of the loop; QuickBASIC jumps to the statement following the key word LOOP. Which, since it's an END statement, stops the program as well. Although the original program did the same thing, this program seems to be more natural.

QuickBASIC allows you to set up a potentially infinite loop at any time—just leave off the tests in a DO-LOOP (an unadorned DO at the top, and an equally unadorned LOOP at the bottom). Once you've done this the loop will end only when QuickBASIC processes an EXIT DO statement (or Ctrl+Break).

You can use the EXIT command to leave a FOR-NEXT loop as well—in this case it takes the form EXIT FOR.

I must confess that I've gotten so used to the way that DO WHILE/UNTIL loops test either at the top or at the bottom that I rarely use the EXIT command to end a loop in the middle. The trouble with ending a loop in the middle is that you can easily lose the sense of what the loop is all about. Our minds seem to work best with loops that end at either the top or bottom. But, as usual, you must be the best judge of what fits your way of thinking.

One word of caution, however. QuickBASIC places no restriction on the number of IF-THEN leading to an EXIT DO statement. It's all too easy to find yourself writing a loop that has 47 different ways to EXIT. It's awfully hard to figure out what this kind of loop is really doing, and it's a nightmare to debug. For this reason some computer scientists advocate that all loops be "single entry single exit" (one way in and only one way out). This would mean using at most one EXIT DO command—and only if you hadn't placed a test at the top or bottom. Like most academic rules this goes too far: rather than have a eureka loop with 10 conditions:

```
DO UNTIL A=3 or B = 4 OR C= 5 OR D$="DOG" . . . .
```

I'd probably use multiple EXIT-LOOP commands. Similarly, you often have to have two ways out of the loop: the first the "natural" way—the way you expected the loop to end—and the second for "un-natural activities"—to bulletproof the loop. Readability and clarity should be the goal, not obeying some abstract rule.

Finally, once in a while you don't want to EXIT the loop—you want to end the program prematurely. For example, suppose you want to give someone three tries to enter a password:

```
' CH2\P9.BAS
' ending a program prematurely

Tries = 0
DO
  IF Tries = 3 THEN END
  PRINT "Password please"
  Password$ = INPUT$(12)
  Tries = Tries + 1
LOOP UNTIL Password$ = "Gary Cornell"
```

Now if someone gives three incorrect passwords then the program ends. Do you see why this fragment gives a person three chances, and not two, or four?

Named constants and more on debugging

As you might expect, ultimately QuickBASIC deals with IF-THEN's and DO loops by converting the relational operators to numbers. So for example in the statement:

```
IF Done THEN PRINT "I'm done!"
```

you'll see the message, if and only if the value of the variable Done was non-zero (= TRUE). An extension of this idea lets you make all your loops single entry/ single exit. Suppose, for example, that you wanted to write a program that would accept a single word. In this example, a word is any sequence of characters ending with either a space, a comma, a semi-colon, a colon, a period, a question mark, an exclamation point or a carriage return (in computerese these are examples of delimiters). At this point such a program shouldn't be hard. You select on the basis of the character and the key line might be:

```
CASE " " , "," ,";" , ":","?","!",CHR$(13)
EXIT DO
```

The trouble with this is that all the commas make this close to unreadable. Instead I'd probably write something like:

```
'CH2\P10.BAS
' A super INPUT - for a single word

CLS
PRINT "This program accepts a single `word' and then echoes it."
Word$ = ""

DO
  Char$ = INPUT$(1)
  SELECT CASE Char$
     CASE " "
       EXIT DO
     CASE ","
       EXIT DO
     CASE ";"
       EXIT DO
     CASE ":"
       EXIT DO
     CASE "."
       EXIT DO
     CASE "!"
       EXIT DO
     CASE "?"
       EXIT DO
     CASE CHR$(13)
       EXIT DO
     CASE ELSE
       PRINT Char$;
       Word$ = Word$ + Char$
   END SELECT
LOOP

PRINT
PRINT "Here is your word again stripped of any punctuation ";
PRINT  Word$

END
```

As I mentioned before, some people don't like the idea of 8 different ways to leave a loop. They prefer to set up a flag variable, and they might re-write the program:

```
'CH2\P10.BAS
' A super INPUT - for a single word
' Revisited

  Word$ = ""
  Done% = 0

DO

  Char$ = INPUT$(1)

  SELECT CASE CHAR$

     CASE " "
       Done% = -1
      CASE  ","
        Done% = -1
      CASE ";"
        Done% = -1
      CASE  ":"
        Done% = -1
      CASE "."
        Done% = -1
      CASE "!"
         Done% = -1
      CASE "?"
         Done% = -1
      CASE CHR$(13)
         Done% = -1
     CASE ELSE
       PRINT Char$;
       Word$ = Word$ + Char$
     END SELECT

LOOP UNTIL DONE%

PRINT
PRINT "Here is your word again stripped of any punctuation ";
PRINT  Word$
END
```

Here the loop ends when the variable Done% flips from 0 (false) to −1 (true)—and this happens when the user enters one of the delimiters. By rewriting the loop this way it now has only one way out—when the variable Done% is true. This is an especially useful technique once you learn about user-defined functions. See chapter 3.

If you end up preferring single entry/single exit loops (and it's clearly a matter of taste), then I suggest you combine it with QuickBASIC's *named constant* feature. This feature lets you give mnemonic names to constants. One problem with the previous program is that all those "Done% = −1" statements are highly non-mnemonic. For example start with:

```
    CONST TRUE = -1
```

or

```
    CONST FALSE = 0
```

or even

```
    CONST FALSE = NOT TRUE
```

So now instead of saying Done% = −1 you can say Done = TRUE, which certainly looks better to my eye.

A named constant consists of the key word CONST followed by any legal QuickBASIC name. You can only set up (assign) to a named constant once. You may have constants of any type. For example:

```
    CONST ErrorMessage = "Bad Mistake!"
```

would set up a string constant. You can add the $ type identifier if you want to but it's not needed. Or,

```
CONST Pi = 3.141592635
```

would give π to 10 places.

Programs that do a lot of Boolean operations almost always start out with the two named constants TRUE and FALSE given as above. Failing to do so often makes the program a morass of -1's and 0's.

Named constants can make your programs more readable in other ways: they prevent the MEGO (My Eyes Glaze Over) syndrome common when reading programs that have lots of mysterious numbers sprinkled about. For example, if you were writing a program that worked with a printer, you might set up a named constant:

```
CONST PaperSize = 66
```

rather than use a variable PaperSize% that you assign the value 66 to. The main advantage of using a named constant over a variable is that you can't change it—even by mistake. Stand-alone programs with named constants also run a bit faster!

In any case, use one or the other. As you can well imagine, programs that work with a printer that say:

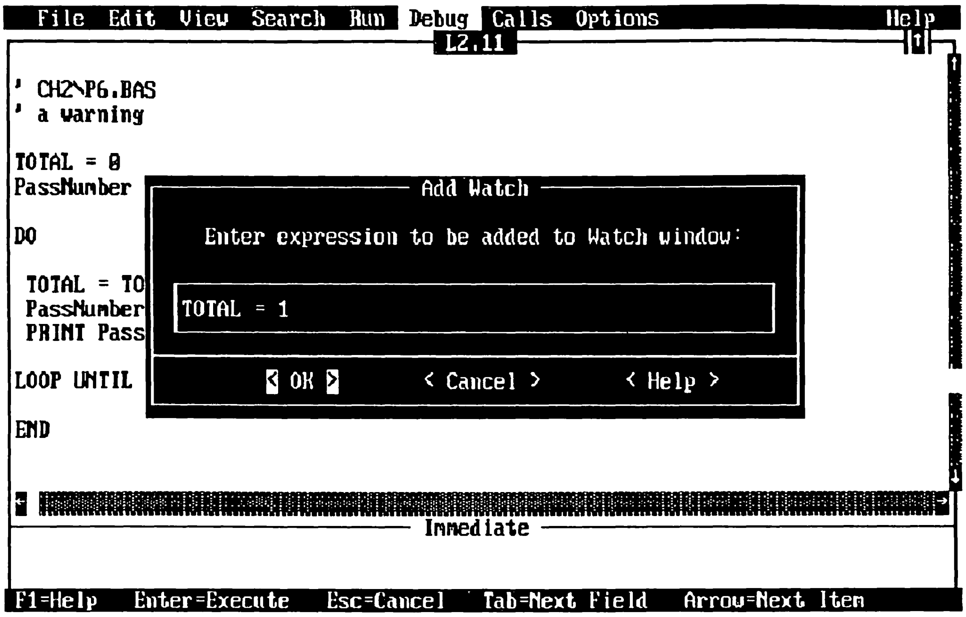
```
DO UNTIL Count = 66
```

are a lot less clear than ones that say:

```
DO UNTIL Count = PaperSize.
```

Suppose you have to debug a program that uses an IF-THEN. Obviously it would be helpful not only to watch the value of a variable but also to "watch" if a relation is true or false. This is done via the Add Watch option on the Debug menu. Although you may have used this instead of the instant watch (SHIFT+F9) to follow the value of a variable it has more power than that. Using it you can watch Boolean relations—any expression that can only be either true or false. For example, look at Fig. 2-7 which comes from the warning program on loops (after all, an expression like TOTAL = 1 is also a Boolean expression). After I hit Enter the watch window will look like Fig. 2-8. Now, by single stepping through the program, possibly using a breakpoint, I can see that this relation stays false. Probably, you'd want to add the variable PassNumber via instant watch.

Think about watching a Boolean variable as a debugging tool, if one of the branches of an IF-THEN never seems to be chosen, is chosen too often, or if a loop never terminates.

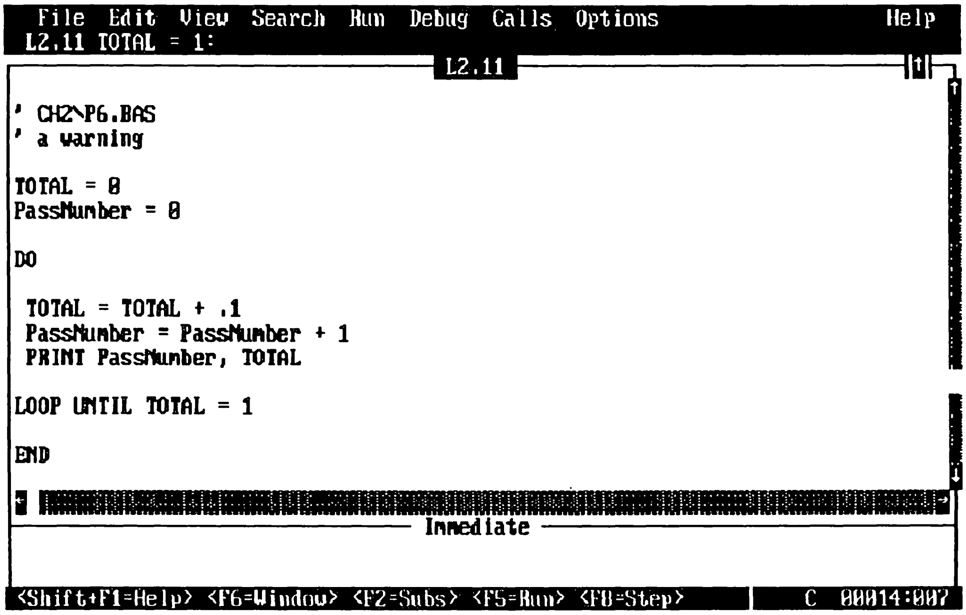

File Edit View Search Run Debug Calls Options Help
L2,11
Add Watch
OK
Cancel
Help
Immediate
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

```
' CH2\P6.BAS
' a warning

TOTAL = 0
PassNumber

DO

 TOTAL = TO
 PassNumber
 PRINT Pass

LOOP UNTIL

END
```

Enter expression to be added to Watch window:

```
TOTAL = 1
```

2-7 Add watch.


File Edit View Search Run Debug Calls Options Help
L2,11
Immediate
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>
C 00014:007

```
L2,11 TOTAL = 1:
```

```
' CH2\P6.BAS
' a warning

TOTAL = 0
PassNumber = 0

DO

 TOTAL = TOTAL + .1
 PassNumber = PassNumber + 1
 PRINT PassNumber, TOTAL

LOOP UNTIL TOTAL = 1

END
```

2-8 The watch window with a Boolean relation.

3

CHAPTER

The built-in functions

This chapter covers QuickBASIC's built-in functions. These commands transform the raw data into the form you need. For example, there are functions that take strings apart as well as ones that put them together. You'll also see how the pseudo-random number generator lets you build an element of indeterminacy into your programs—a necessary tool for programming games of chance or simulations.

Finally, as always, be prepared to check the on line help about specific functions—the examples given there complement the ones given here.

Simple string functions

Suppose you wanted to center a message on the screen. You can use the LOCATE command to move the cursor, but where exactly should you move it to? For example, suppose you had the message "HELLO WORLD!". This is 12 characters long, counting the space. To center this string on a line you have to move back from the midpoint six positions, and only then start printing. In general, you need a command that finds the length of a string. In QuickBASIC it is: LEN( ), where the parentheses hold a string expression. Here is a program to center a message on the screen.

```
' CH3\P1.BAS
' centering a message on the screen

CLS

PRINT "Enter a string to center on the screen.  Press return"
PRINT " when done."
LINE INPUT "?"; Message$                    'A
LenMessage = LEN(Message$)

  IF LenMessage > 80 THEN                   'B
    PRINT "TOO LONG!"
    BEEP
  ELSE
    CLS
    LOCATE 12, 40 - (LenMessage / 2)        'C
    PRINT Message$
  END IF

END
```

A) I use the LINE INPUT command because there may be a comma within the message. And, as usual the "?" is there because the LINE INPUT command doesn't display one.
B) Notice how the "block IF-THEN-ELSE" makes the program cleaner. When the message is too long, the second statement provides some audio reinforcement. Otherwise it processes the ELSE block that centers the message.
C) If the message has odd length then taking half its length gives a fraction. But, as you might expect, since the LOCATE command expects an integer, it rounds off any fraction.

Finally, notice how, at the beginning of the program, I calculated the length of the message by setting up a new variable. Obviously, as programs grow more complicated this technique will save even more time. It's silly to recalculate a constant in each clause of an IF-THEN.

The + command lets you join two strings together to form a new string. Since the limit on a single string is 32,767 characters, you usually can combine strings to your heart's content. However, you're more likely to turn to the commands in QuickBASIC that cut up strings. These let you pull individual letters or larger chunks out of a string. The most important of these functions is:

```
MID$ ( ,        ,        ,)
```

The first entry holds the string (or string expression) that you want to cut up. Next comes the starting position of the characters that you want cut out of the string.

The last position specifies the number of characters that you want to pull out. Examples:

```
MID$("Quick BASIC",1,5) = "Quick"
MID$("Quick BASIC",1,6) = "Quick "        '(note the space!)
MID$("Quick BASIC",7,5) = "BASIC"
MID$("Quick BASIC",6,5) = " BASI"
MID$("Quick BASIC",6,6) = " BASIC"        '(note the space!)
```

If you leave out the last entry—the one telling how many letters to pull out—then QuickBASIC retrieves the rest of the string, starting from the position determined by the second entry. So:

```
MID$("QuickBASIC",7,5) = MID$("QuickBASIC",7) = "BASIC"
```

You also get the rest of the string if the third entry is too large—greater than the number of characters remaining.

The jargon is to say that MID$ is a function of three (or occasionally two) arguments. Argument is a term borrowed from mathematics. Think of it as meaning "pieces of information massaged." The MID$ usually uses three pieces of information—a string in the first position and integers in the remaining two positions.

Suppose you want to "explode" a word by printing it vertically instead of horizontally. The program must grab a letter from the word, print it, and then move on to the next letter in the word. Obviously, this calls for a FOR-NEXT loop. Here is a program for this:

```
' CH3\P2.BAS
' exploding a word

DEFINT I
CLS

INPUT "Enter a string to explode"; Message$
LenMessage = LEN(Message$)                    'A

   FOR I = 1 TO LenMessage
     PRINT MID$(Message$, I, 1)               'B
   NEXT I

END
```

A) Since the limit for the FOR-NEXT loop doesn't change (its a loop invariant) it should be computed once—before the FOR-NEXT loop starts.
B) On each pass through the loop, the position (the value of I) where the MID$ command starts working increases. The number of characters pulled out remains the same—one.

If you change the MID$ command slightly, the program works quite differently. For example, change a crucial line in it to read:

```
PRINT MID$(Message$,1,I)
```

and now, on each pass through the loop, the program pulls out more and more letters (the third position controls the number of letters pulled out). The result is a "word ladder," i.e., if

```
Message$ = "QuickBASIC"
```

then you'll see:

```
Q
Qu
Qui
Quic
Quick
etc.
```

Similarly, if in the original program you have the loop go backwards (FOR LenMessage TO 1 STEP −1) then the program explodes the word backwards. Add a semi-colon to the PRINT statement and the program will print the word backwards.

You can use this to check if a phrase is a palindrome. A palindrome reads the same backwards as forwards—spaces and punctuation not counting. A strong palindrome is one where the spaces and punctuation do count. "MADAM I'M ADAM" is a weak palindrome, while "ABLE WAS I ERE I SAW ELBA" is a strong palindrome. Writing a program to check if a phrase is a strong palindrome is easy, using MID$. Here is a naive version:

```
' CH3\P3.BAS
' a `strong' palindrome checker

DEFINT I
CLS

INPUT "Enter a string to check"; Message$
LenMessage = LEN(Message$)
Reverse$ = ""

   FOR I = LenMessage TO 1 STEP -1
     Reverse$ = Reverse$ + MID$(Message$, I, 1)    'A
   NEXT I

IF Reverse$ = Message$ THEN
    PRINT "Success! "; Message$; " is a strong palindrome."
  ELSE
    PRINT "No luck."
 END IF

END
```

A naive approach to solving a problem is usually a bit inefficient, and this program is no exception. The inefficiency comes because we build up the entire reversed phrase before checking if we have a palindrome.

How can this program be made more efficient? How can it be modified to check for any kind of palindrome? As usual, efficiency requires a more complex program and the possibilities for mistakes grow.

Famous dialogue between two programmers:

Young hotshot programmer: "My program is more complex than yours, but it will theoretically run twice as fast."

Experienced programmer: "That may be true, but your program doesn't work and mine does."

One outline for a more efficient program would be:

(1) Find the next character going forward

(2) Find the next character going backward

If they are equal and there are characters left, repeat steps (1) and (2). If the loop stopped because two characters aren't equal, then the phrase isn't a strong palindrome. If the loop stopped because of "no more characters," then the phrase is a strong palindrome.

One nice feature of this outline is that it can be modified to check for a weak palindrome. Change (1) and (2) to read: "Find the next alphabetic character . . .".

More on strings

The MID$ command has two cousins that are occasionally useful: LEFT$ and RIGHT$. As the names suggest, LEFT$ picks out characters from the beginning of a word and RIGHT$ picks them out from the end. Of the two, RIGHT$ is the more common. It avoids a subtraction inside the MID$ command and can work a bit faster as a result. For example:

```
MID$(A$,LEN(A$)-3,3)  MID$(A$,LEN(A$)    -    3)  RIGHT$(A$,3)
```

all have the same effect.

LEFT$ works the same way, but only saves you from putting a one in the second position in the MID$ function. If you want the first five characters in a string use either:

```
MID$(A$,1,5)  or  LEFT$(A$,5)
```

QuickBASIC has around 100 built-in functions—and many are there to maintain compatibility with interpreted BASIC. It shouldn't be surprising, then, that many of them overlap in their powers.

When you press a function or arrow key, the INPUT$(1) command only shows that some key was pressed; but you can't always use it to tell which one. To determine which one of these non-ASCII coded keys was pressed, you combine the INKEY$ command with the string manipulation functions you've just seen. Suppose you set KeyPress$ = INKEY; then:

IF LEN(KeyPress$) = 0 then no key was pressed.

IF LEN(KeyPress$) = 1 then a key or key combination having an ASCII code was pressed. Here, as you know, the value of the variable KeyPress$ is the key pressed.

IF LEN(KeyPress$) = 2 then a non-ASCII coded key was pressed. In this case the first character has ASCII value 0 and the ASCII code of the second character determines which key was pressed. To pick up this code is rather messy. Since it's the code given by the right most character that matters, you use:

```
ASC(RIGHT$(KeyPress$,1) )
```

Here's a table that tells you what the most common of these code numbers mean. A complete table of these scan codes is available via the online help facility.

Code For Second Character	Key Pressed
16 - 25	Alt/Q-ALT/P (2nd row + Alt)
30 - 38	Alt/A-Alt/L (3rd row + Alt)
44 - 50	Alt/Z-Alt/M (4th row + Alt)
71	HOME
72	UP ARROW
73	PAGE UP
75	LEFT ARROW
77	RIGHT ARROW
79	END
80	DOWN ARROW
81	PAGE DOWN
82	Ins
83	Del

In other words: if someone presses the Ins key, then RIGHT$(KeyPress$,1) = CHR$(82).

Now that you know these codes it's easy to modify the cursor mover program from chapter 2.

```
'CH3\P4.BAS
' A cursor mover revisited

CLS
KEY OFF
LOCATE 1, 1, 1, 0, 7              ' make the cursor large

Row = 1:Col = 1
```

```
DO
   LOCATE Row, Col

   DO
    KeyPress$ = INKEY$
   LOOP WHILE KeyPress$ = ""

KeyStroke$ = RIGHT$(KeyPress$, 1)              'A

 SELECT CASE KeyStroke$

    CASE CHR$(4), CHR$(77)          'B  - CNT/D or Right Arrow_
      IF Col <= 79 THEN Col = Col + 1

    CASE CHR$(19), CHR$(75)           'CNT/S or Down Arrow
         IF Col > 1 THEN Col = Col - 1

    CASE CHR$(5), CHR$(72)                'CNT/E  or Up Arrow
      IF Row > 1 THEN Row = Row - 1

    CASE CHR$(24), CHR$(80)               'CNT/X or Down Arrow
      IF Row < 25 THEN Row = Row + 1

    CASE CHR$(18), CHR$(73)                'CNT/R or Pg Up
         Row = 1

    CASE CHR$(3), CHR$(81)                  'CNT/C or Pg Down
         Row = 25

    CASE CHR$(71)                           ' C  Home
       Row = 1
       Col = 1

     CASE ELSE
       IF KeyStroke$ <> "X" THEN BEEP
    END SELECT


 LOOP UNTIL KeyStroke$ = "X"

LOCATE 1, 1, 1, 7                ' cursor back to normal
END
```

A) If the key pressed is a normal ASCII coded key then it has length one—and RIGHT$(,1) is a fancy way to get the key. If it has length two then, as mentioned before, this gives the character whose code we need to check.

B) Notice how simple it is to modify the SELECT CASE statement from the previous program to this new situation. In each case all that's needed is to add a check for an extra character.

C) I added a Home cursor option—I'll leave adding an "END option" to you.

The MID$ command has one other useful feature. It lets you make changes inside a string. For example, if:

```
BestBASIC$ = "TurboBASIC"
```

then the statement:

```
MID$(BestBASIC$",1,5) = "Quick"
```

gives BestBASIC$ the value "QuickBASIC". When you use MID$ this way, the second position controls where the change will start and the third position controls how many letters to pull out from the string on the right hand side. These are the letters that will be switched into the original string. For example,

```
MID$(BestBASIC$",1,5) = "QuickBASIC by MicroSoft"
```

gives the same result as before. If the right hand side has fewer characters than the entry in the third position of the left hand side demands, then QuickBASIC changes as many characters as occur on the right hand side. So:

```
MID$(BestBASIC$",1,5) = "QB"
```

gives BestBASIC$ the value "QBrboBASIC."

The MID$ statement makes changes within a string but never changes the length of the original string. If the number in the third position is too large relative to the number in the second position—greater than the remaining number of characters—then only the characters remaining can change. Finally, just as with the MID$ function, you can leave out the last position:

```
MID$("In the beginning ",8) = "middle was"
```

changes the string to:

```
"In the middle was"
```

In this case there's just enough room to fit the string on the right hand side into the string on the left—starting at the eighth position. (Counting from the eighth position there are 10 characters left in the phrase "In the beginning was"—the space counts.)

If you want to change the size of a string then the MID$ statement is of little use. This requires brute force—it's a bit like splicing tape. For example, suppose we want to change the string:

"Turbo Pascal is the best programming language"

to read:

"QuickBASIC is the best programming language."

Since the string "Pascal" has 6 letters and the string "BASIC" has five, you cannot use the MID$ statement. Instead you must follow the splicing analogy:

Cut out the phrase "Turbo Pascal"

Hold the phrase "is the best programming language"

Splice in the phrase "QuickBASIC," and reassemble.

Here's the fragment:

```
phrase$ = "Turbo Pascal is the best programming language"
Begin$ = "QuickBASIC"
EndPhrase$ = MID$(phrase$, 13)
phrase$ = Begin$ + EndPhrase$
PRINT phrase$
```

Programming this kind of change can be a bit painful—if you always have to go in and count characters in order to find which position, for example the word "Pascal," started (and ended). This task is simplified by another of QuickBASIC's built-in functions: INSTR. Like the MID$ function, INSTR also works with three (and occasionally two) pieces of information—it's a function of three (occasionally two) arguments.

INSTR tells you whether a string is part of another string or not (in computerese: is a substring of); and, if it is, at what position the substring starts. Using the same variable Phrase$ as above, then the value of:

```
INSTR(1,Phrase$,"Pascal")
```

is 6, because the string "Pascal" occurs in the phrase "Turbo Pascal is the best programming language," starting at the sixth position.

In this case, QuickBASIC searches the string, starting from the first position, until it finds the sub-string. If it doesn't find the string then it gives back a value of zero. So:

```
INSTR(1,Phrase$,"pascal")
```

has value 0 because "pascal" isn't a sub-string of the phrase "Turbo Pascal is the best programming language." Remember: case is important inside quotes, and so the INSTR function is case sensitive.

In this situation, the INSTR function essentially translates the following outline into a single command:

Pull out characters 1—6.

Check if they are the searched for substring. If so then return value = 1 and stop.

Now do it for characters 2 through 7. If successful, return value = 2 and stop.

Continue until there are no more sub-strings to check. If totally unsuccessful, return a value of zero.

You can easily translate this outline into a program that, using MID$, would work for any string (replace "5" by the LEN(string))—it's a routine exercise. The program you end up with will, however, take much longer to do what INSTR does.

The general form of the INSTR function is:

INSTR ([where to start,]string to search, string to find,)

Here, I am following the conventions in the QuickBASIC manual and the on-line help: Anything that is optional in the description of a command is enclosed by brackets. In this case, the optional first position specifies from what position to start the search. If you leave this entry out then the search automatically starts from the first position.

Notice as well the comma within the brackets. This means that if you put something here, then you need the comma. If the bracket fell before the comma then this would indicate that the comma is always needed. It's worth getting comfortable with the notations used in the manual and the on-line help. (Remember the online help on Syntax Notation Convention available from the contents hyperlink?)

The deeper you go into QuickBASIC, the more benefit you'll derive from the manual and the online help. In particular, as I mentioned in the introduction to this chapter, the examples in the QuickBASIC manual and available on line can't help but complement the explanations I give here.

Another good example of how to use the INSTR command is a program that parses (computerese for take apart) a string, say a name, into its component parts. Suppose you had a string made up of individual words—each separated from the next by a single space. Then a naive outline to pull out the individual words is:

Find the first occurrence of a space. The stuff up to the first space is a word.

Find the second space. The stuff between these two spaces is a word.

Now find the third space. The stuff between the second and third space is a word.

Continue until there are no more spaces.

Now the stuff between the last space and the end of the word is the last word. Here is a program that implements this outline:

```
'CH3\P5.BAS
' This program uses INSTR to parse a phrase
' by searching for a space as the separator

CLS

DO                                                    'A
 INPUT "Please enter the phrase to parse"; Phrase$
 LenPhrase = LEN(Phrase$)
LOOP UNTIL LenPhrase > 0

Separate$ = CHR$(32)             '= space

BeforeSpace = 0                                       'B
AfterSpace = INSTR(BeforeSpace + 1, Phrase$, Separate$)    'C

DO UNTIL AfterSpace = 0                               'D
```

```
  SizeOfWord = AfterSpace - BeforeSpace - 1             'E
  NextWord$ = MID$(Phrase$, BeforeSpace + 1, SizeOfWord)   'F
  PRINT NextWord$
  BeforeSpace = AfterSpace                               'G
  AfterSpace = INSTR(BeforeSpace + 1, Phrase$, Separate$)   'H

 LOOP

   PRINT MID$(Phrase$, BeforeSpace + 1)                   'I

END
```

A) It's hard to take apart the null string.
B) This initialization makes it easy to start the process. Without this the program would have to treat the first word separately.
C) Get the next space by starting one position in from the previous space. In this case from the first position (because of A).
D) The loop stops when no more spaces are encountered. By testing at the top we take care of the case when there is only a single word. See I).
E) I figured this out by working an example through on paper. For example, if spaces were at the 5th and 9th positions, then the actual word took up positions 6,7, and 8, i.e., was three characters long.
F) Computing the size of the word in the previous step made this statement cleaner. There's rarely a need to combine many statements into one.
G) This moves us along to the position of the next space—in preparation for;
H) Looking for the next space. If we find it, the cycle continues, if not we move to;
I) There are no more spaces—the rest of the string must be a word.

One problem with this program is that it doesn't handle multiple spaces within the phrase very well (actually, at all). The problem, as usual, stems from the outline—I assumed that a word is bounded by no more than one space. How can we take care of the quite common possibility of a double space, or even more? Changing the outline is simple—we only have to, for example, change:

"Find the second space. The stuff between these two spaces is a word."

to read;

"Find the second space. The stuff between these two spaces is a word—if it is not empty space."

But what really happens if there are two consecutive spaces? As always, an example helps. Suppose the Phrase$ was:

"THIS IS   A TEST"

(In QuickBASIC:

"THIS" + SPACE(2) + "IS" + SPACE(3) + "A" + SPACE(1) + "TEST").

Let's "play computer." Especially when dealing with loops this is often best done by setting up a little chart detailing the value of the variables on each pass. You often use a table like this when you debug with the instant watch—WATCHing is of little use unless you know what's supposed to happen:

PASS NUMBER	VALUE of (key) variables
0 (before)	BeforeSpace = 0
entering loop)	AfterSpace = 5
(start of) 1st	SizeOfWord = 4
loop.	
(end of) 1st	BeforeSpace = 5
pass	AfterSpace = 6
(start of) 2nd	SizeOfWord = 0
pass	
(end of) 2nd	BeforeSpace = 6
pass	AfterSpace = 9
(start of) 3rd	SizeOfWord = 2
pass	
(end of) 3rd	BeforeSpace = 9
pass	AfterSpace = 10
(start of) 4th	SizeOfWord = 0
loop.	

As you can see, this little table seems to indicate that whenever we have two spaces together the variable SizeOfWord has value zero. Does this always have to be true? Yes, it does, because if you have two consecutive spaces then the value of AfterSpace is always one more than the value of BeforeSpace. And so the value of:

AfterSpace − BeforeSpace − 1 = SizeOfWord

must be 0.

Knowing this makes it easy to modify the program. (Change the PRINT statements to read: IF SizeOfWord > 0 THEN . . .) I'll leave the changes to you. A version of this program that takes this into account and also checks for other word separators, like commas, periods and question marks can be found in the next chapter.

Finally, although you need a bit of work to pull out extra spaces within a string, you don't have to do very much for spaces at the beginning or end of a string. QuickBASIC comes with two built-in functions for this: LTRIM$ ( ) and RTRIM$ ( ). As the names suggest, these functions trim spaces from the left and right respectively. For example, if:

A$ = "   This has too many spaces on the left."

then:

LTRIM$(A$) = "This has too many spaces on the left."

or

```
    A$ = "This has too many spaces on the right.     "
```

then:

```
    RTRIM$(A$) = "This has too many spaces on the right."
```

Manipulating numbers as strings

Because the string manipulation functions are so powerful, QuickBASIC gives you a way to change a number into a string of digits. The command STR$ does this: IF

```
X=1234567890
```

then

```
STR$(X) =" 1234567890"
```

Notice the space. QuickBASIC always adds an extra space at the beginning of the string representation of a positive number for the implied plus sign. Converted negative numbers keep the minus sign, and have no space at the beginning. For example, the following fragment would print a number without any spaces:

```
X$ = STR$(A)
PRINT LTRIM$(X$).
```

At first glance the STR$ command does not seem very useful. For example, PRINT USING can do the same thing as the above fragment. Moreover, when you change a number into a string you lose the ability to add, subtract, or multiply. What you gain though in return is the ability to isolate each digit. After you isolate a digit you can use another function—the VAL function—to change that digit back into a number.

Here's an example of why this might be useful. One problem when using the PRINT USING command is to decide on the number of digits before the decimal point. If you are wrong, then you'll be stuck with the % flag in front of the number. This is easy to do if you combine the STR$ command with the INSTR command. Here's an outline for this:

Change the number to a string.

Find the decimal point using INSTR.

If the value from the previous step is zero then there's no decimal point and the number of digits is one less than the length of the converted number. Otherwise, the number of digits in front of the decimal point is two less (because of the extra space that STR$ sticks on and the decimal point) than the value given by the INSTR function.

Here's the QuickBASIC fragment:

```
' find the number of digits
    Number$ = STR$(Number)
```

```
Digits = INSTR(Number$,".")
IF Digits = 0 THEN
    NumOfDigits = LEN(Number$) - 1
ELSE
    NumOfDigits = Digits - 2
END IF
```

As another example of the power of STR$: shipping companies often determine the charge for sending a package by looking at the first three digits of its zip code. For example, they might use the following table to determine the zone:

First 3 digits of Zip Code	Zone
001 TO 374	8
375 TO 399	7
500 TO 599	5
600 TO 699	4
700 TO 799	3
800 TO 999	2

Once we isolate the first three digits of the zip code, then we can use the VAL command to change it back into a number. Once it's a number then we can use the SELECT CASE statement to check the range. Here is a program that does exactly this:

```
'CH3\P6.BAS
' A simple zip code check
' uses STR$ and VAL

CLS

INPUT "Enter the zip code for the address"; ZipCode

ZipCode$ = STR$(ZipCode)                              'A
Zone$ = MID$(ZipCode$, 2, 3)                          'B
Zone = VAL(Zone$)                                     'C

LOCATE 12, 37

 SELECT CASE Zone

   CASE 1 TO 374
    PRINT "Zone 8"

   CASE 375 TO 399
    PRINT "Zone 7"

   CASE 500 TO 599
    PRINT "Zone 5"
```

```
  CASE 600 TO 699
   PRINT "Zone 4"

  CASE 700 TO 799
   PRINT "Zone 3"

  CASE 800 TO 999
   PRINT "Zone 2"

  CASE ELSE
   PRINT "Not a zip code"

 END SELECT

END
```

A) Here, as before, the string command changes a zip-code into a string of digits.
B) Now we isolate the first three digits of the zip code. Note we have to start at the second position because of the extra space that QuickBASIC sticks on in a transformation using STR$.
C) Now change it back into a number so that QuickBASIC can check its range.

Note that the VAL command disregards any spaces at the beginning of a string of digits. However, keep in mind that although the VAL command does disregard any spaces in the front of a word, it reacts badly to other non-numerals in the word. For example, if the leading character of a string isn't a numeral then VAL always gives 0. Moreover, if any intermediate character isn't a space, then the conversion stops at that point, for example, VAL("12zx345") has the value 12, but VAL("12   345") has the value 123450.

Another use of VAL is to give you a super INPUT command. Suppose you want part of a program to accept a number but disregard commas, extraneous decimal points, etc. The outline for this is:

Get a character.

If it's a digit, stick it on the right.

If it's the first decimal point, accept that too and also stick it on the right. Otherwise disregard it.

Continue until the return key is hit. Now change the string into a number. Here is a program that follows this outline:

```
'CH3\P7.BAS
' This program combines VAL
' and INPUT$ to accept a number and only a number

CLS
PRINT "Type a number - commas allowed!  Hit enter when done."
```

```
CONST TRUE = -1                              ' - 1 is true
CONST FALSE = 0

Numeral$ = ""
DecimalFlag% = TRUE                          'A

 DO
   KeyStroke$ = INPUT$(1)                    'B

   SELECT CASE KeyStroke$

     CASE "0" TO "9"                         'C
       Numeral$ = Numeral$ + KeyStroke$
       PRINT KeyStroke$;

     CASE "."

        IF DecimalFlag% THEN                  'D
          Numeral$ = Numeral$ + KeyStroke$
          PRINT KeyStroke$;
          DecimalFlag% = FALSE
       ELSE
          BEEP                                'E
       END IF

     CASE ","
       PRINT KeyStroke$;

     CASE CHR$(13)
        PRINT KeyStroke$                      ' F

     CASE ELSE
       BEEP                                    'G

    END SELECT

  LOOP UNTIL KeyStroke$ = CHR$(13)

PRINT "The number was "; VAL(Numeral$)              'H

END
```

A) In order to make the program more readable, I've used the named constants that you saw in the previous chapter.

B) A loop using INKEY$ would work as well. With both these commands, however, the character isn't displayed (echoed). This means that the program has to display the digits as they are entered—otherwise, there's no feedback to the user.

C) This is the key to this improved INPUT routine. If a character is in this range, then it's a digit. So we concatenate it at the right end of the string of numerals that we will eventually transform into a number. Also, as mentioned above, we have to display it (as usual, the semi-colon in the

PRINT command suppresses the automatic carriage return) in order to provide some visual feedback to the user.

D) This case accepts a single decimal point in the number. However, entering a decimal point flips the "flag" to false (=0).

E) If this case is selected after the variable, DecimalFlag% is set to 0, then the ELSE clause is always processed—and a beep follows.

F) Entering a carriage return lands the program in this case. This is the case that ends the DO-LOOP. Printing it here gives a blank line before the number is displayed in H. You also could have used the EXIT LOOP here instead and eliminated the test at the bottom.

G) Audio feedback is a good way to make programs more user friendly.

H) This statement displays the number. Obviously if this program was part of a larger program we would probably manipulate it in some way at this point.

The RND function

There are programs that let you play blackjack or backgammon or most any game of chance with a computer. In card games and other games, the play is unpredictable—this is exactly what is meant by a game of chance. On the other hand, computers are machines and the behavior of machines should be predictable. To write a program in QuickBASIC that allows you, for example, to simulate the throwing of a die, you need a command that makes the behavior of the computer seem random. This is done by means of the command RND. Look at the following program:

```
' CH3\P8.BAS
' 10 pseudo-random numbers

CLS

FOR I = 1 TO 10
 PRINT RND(1)
NEXT I

END
```

After you compile and run this program, ten numbers between 0 and 1, each usually having 16 digits, roll down the screen. These numbers seem to follow no pattern: that's what we usually mean by random. They'll also have many but not all of the sophisticated statistical properties that scientists expects of "random" numbers. Each time the computer processes a line containing the statement PRINT RND(1), a different number between 0 and 1 pops out. In theory the number can be 0 but can't ever be 1.

It's natural to wonder what a number with 16 decimal places is good for. Suppose, for example, we wanted to write a program that simulated a coin toss. There are three possibilities: it could be heads, it could be tails—or it could stand on edge (don't wait up for this to happen). A program to simulate a coin toss might look like this:

```
' CH3\P9.BAS
' A coin toss simulator

CoinToss = RND

  SELECT CASE CoinToss

   CASE IS < .5
      PRINT "Heads"
   CASE .5
      PRINT "Stood on edge!!!!"
   CASE ELSE
       PRINT "Tails"

  END SELECT

END
```

The next program varies this by keeping track of the number of different kinds of flips:

```
' CH3\P10.BAS
' A multiple coin toss simulator

DEFINT I-Z
CLS
INPUT "How many trials"; Trials

NumOfHeads = 0
NumOfTails = 0
Unbelievable = 0

FOR I = 1 TO Trials

  CoinToss = RND(1)                              'A

  SELECT CASE CoinToss

   CASE IS < .5
      NumOfHeads = NumOfHeads + 1
   CASE .5
      PRINT "Stood on edge!!!!"    'maybe in a couple of years?
      BEEP: BEEP
      Unbelievable = Unbelievable + 1
   CASE ELSE
      NumOfTails = NumOfTails + 1

  END SELECT
```

```
NEXT I

PRINT "Number of heads was"; NumOfHeads
PRINT "Number of tails was"; NumOfTails

IF Unbelievable > 0 THEN
  PRINT "The coin stood `on edge'!"
END IF

END
```

Notice that because of the DEFINT I-Z command I could not choose a variable whose name began with a letter in this range. If I had called the variable, say Toss instead of CoinToss, then it would always be an integer—it would always have the value 0 or 1 depending on how it was rounded off. Since it's equally likely that the number would be rounded up as rounded down, I could have changed the program to determine the coin's position using this—if I didn't mind eliminating the "standing on edge" possibility (which to be quite honest has never happened, no matter how many trials I've run).

Try this program with different number of trials. You should get roughly the same number of heads as tails and no "standing on edges." For a large number of trials it would be very unlikely that you'd get equal numbers of heads and tails. Now run this program using the same number of trials each time. If you do, then you'll notice that you get exactly the same number of heads and tails. This would certainly be unusual behavior for an honest coin. What gives?

Well, the numbers you get using RND(1) are only pseudo-random. "Pseudo" generally means false, and you've just seen one of the problems of pseudo-random numbers. Every time you start a program anew then you will get the same sequence of pseudo-random numbers. It's as if there is a book of these numbers in the computer's memory and after each program is over the book gets turned back to page one. So the deck always starts out in the same order and therefore the results are fixed. We need a way to "shuffle the deck" each time the program starts up. This can be done in many ways. A slow but sure way is to start your program with something like:

```
PRINT "Everyone who wants to play should press a number key."
PRINT "Then hit the return key."
INPUT Number%
    FOR I% = 1 TO Number%
        X = RND (1)
    NEXT I%
```

This fragment advances you to a place in the list that no one can tell beforehand. The problem with this approach is that moving ahead in the list (generating all those pseudo-random numbers) takes time, and it's silly to wait. You can eliminate any wait by changing what you put inside the parentheses of the RND function or slightly modifying it.

First off, suppose you issue a RND(0). Then you will get the last pseudo-random number generated. This is useful in trying to debug a program. RND(0) gives you a way of checking what pseudo-random number the machine just used. Imagine trying to check a program if an important number changes each time you run it and you don't know what it was. From this point on I'm going to be slightly imprecise and stop saying "pseudo-random" and speak of the numbers coming from the RND function as being random.

Suppose next there is a negative number inside the parentheses. Each time you give the command

```
RND(negative number)
```

then you get the same random number. This is another important debugging tool. It lets you re-run a program keeping the random numbers temporarily stable. A good way to think about what a negative seed does is to imagine that there is a different list of random numbers, each one corresponding to a different negative seed.

The best way not to stack the cards is to use the exact time of the system clock to reseed the random number generator. Since the clock is accurate to around a tenth of a second it's quite unlikely that a program will start at exactly the same moment each time it is run. This is done by combining a new command RANDOMIZE with the TIMER command in the form of the following statement:

```
RANDOMIZE TIMER.
```

You saw the TIMER command in chapter 2. Recall that it gives you the number of seconds since midnight—within 1/10 of a second. The RANDOMIZE command is a general shuffler. In this case it takes the value of the TIMER function and uses it to re-seed the random number generator. It can be also used in the form:

```
RANDOMIZE
```

whereupon the program stops and asks for a seed, so you can use the fragment:

```
PRINT "At the ? enter a number."
RANDOMIZE
```

What a user will see is the message: "Random Number Seed?"

Think of a seed as the number from which the random numbers grow. This is done by transforming the seed by a rather complicated method called the linear congruential method. The RANDOMIZE statement can also be a useful debugging tool. This is because you can use any numeric expression in the RANDOMIZE command (RANDOMIZE Xexpress, RANDOMIZE Yexpress) and you get the same random numbers whenever Xexpress and Yexpress have the same value.

In the current version of QB, it's best to use negative numbers as seeds if you want to take advantage of this.

Numbers between 0 and 1 might, with a little work, be good for imitating a coin toss, but the method we used would be cumbersome for, say, a dice simulation. The outline would be something like this:

If the random number is less than 1/6 make it a 1

If less than 2/6 (=1/3) make it a 2

If less than 3/6 a 3

and so on.

Thinking about this outline leads to a simple trick called *scaling* that more or less automates this process. Suppose you take a number between 0 and 1 and multiply it by 6. If it was less than 1/6 to start with it will now be less than 1, if it was between 1/6 and 2/6 (1/3) it will now be between 1 and 2, and so on. All we need to do then is follow the outline.

Multiply the number by 6 and move up to the next integer. In general, if the number was between 0 and 1 (but never quite getting to 1), then the result of multiplying by 6 goes from 0 not quite up to 6.

Unfortunately, there's no command in QuickBASIC to move up to the next integer. Instead there's a command to throw away the decimal part of a number. It's FIX; for example:

```
FIX(3.456)    = 3  FIX(-7.9998 = -7  FIX(8) = 8
```

However, by adding one to the results of FIXing a positive number we will, in effect, move to the next highest positive integer. For example, look at this:

```
'CH3\P11.BAS
' A dice simulation using FIX

RANDOMIZE TIMER
CLS

DIE = FIX(6 * RND(1)) + 1              'A
PRINT "I rolled a"; DIE

END
```

A) This is the key to the program. The number inside the parentheses (6*RND(1)) is always between 0 and 6—but it can't be 6, because RND(1) is never 1. Applying the FIX function gives you an integer between 0 and 5 (i.e., 0,1,2,3,4, or 5) and now we only have to add one to make it a proper looking die.

There's another function that works much the same way as FIX; it's INT. INT gives the "floor" of a number, the first integer that's smaller than or equal to the number (it's technically called the greatest integer function). Thinking of it as the floor function makes it easy to remember what happens for negative numbers. With negative numbers you move down. For example, INT(−3.5) is −4, INT(−4.1) is −5 and so on. So FIX and INT work the same way for positive numbers, but are different for negative ones. Using INT and adding one always moves to the next largest integer. Here is a rewritten version of the previous program using INT—it looks much the same:

```
'CH3\P12.BAS
' A dice simulation using INT

RANDOMIZE TIMER
CLS

DIE = INT(6 * RND(1)) + 1          'see A above
PRINT "I rolled a"; DIE

END
```

The INT and FIX functions have other uses. For example, the post office charges for first class mail is 29 cents for the first ounce and 23 cents for each additional ounce, or fraction thereof. Suppose an item weighed 3.4 ounces; then the cost would be 29 cents for the first ounce and 69 (3*23) for the additional ounces—counting the fraction. The cost is:

```
.29 + INT(3.4)*.23.
```

In general it's:

```
IF INT(WeightOfObject) = WeightOfObject THEN
    COST = .29 + .23*(WeightOfObject - 1)
ELSE
    COST = .29 + 23*(WeightOfObject)
ENDIF
```

More on using the random number generator

Suppose you wanted to write a Jumble program. This would take a "string" and shuffle the letters around. It's a prototype for many other types of operations, for example shuffling a deck of cards. Here's an outline for one way to do it:

Start at the first character.
"Swap" it with a randomly chosen character.
Do the same for the second character.
Until there are no more characters left.

The swapping can be done with the MID$ statement, since you are never changing the size of the string. Here is a program that implements this outline:

```
'CH3\P13.BAS
' a Jumble program
' demonstrates MID$ as a statement
' and the RND function

DEFINT A-Z

CLS
RANDOMIZE TIMER

PRINT "Enter the phrase to jumble."
LINE INPUT "?"; Phrase$
Jumble$ = Phrase$

LenPhrase = LEN(Phrase$)

FOR I = 1 TO LenPhrase

  INum = INT(LenPhrase * RND(1)) + 1                    'A
  NewChar$ = MID$(Jumble$, INum, 1)                     'B
  OldChar$ = MID$(Jumble$, I, 1)                        'C

  MID$(Jumble$, I, 1) = NewChar$                        'D
  MID$(Jumble$, INum, 1) = OldChar$                     'E

NEXT I

CLS
PRINT Phrase$: PRINT
PRINT "In jumbled form: "; Jumble$

END
```

A) This gives you the random position within the string.
B) This gives the character at the random position.
C) This is the character that you are going to swap with the character determined by B. It's determined by the counter in the FOR-NEXT loop.
D) The MID$ statement lets you replace a character within a string, but this statement must be combined with the next statement in order to preserve all the letters in the original string.
E) Since there is no statement like:

```
SWAP MID$(Jumble$,INum,1), MID$(Jumble$,I,1)
```

statements (C), (D), (E) combine to give one.

To make this program into a card shuffler all you need to do is give Phrase$ the right value. You can do this using the CHR$ command. I'll return to this in the next two chapters.

To this point all the random integers that you've used have started from 0 or 1. Sometimes it's convenient to have random integers that span a range. For example, take a random four letter combination, how likely is it to be a word in English? To try this out you need to generate four random letters and string them together. An obvious way to do this is to apply the CHR$ command to a random integer between 65 and 90 (the range of ASCII codes for the uppercase alphabet). To get a random integer in this range:

    Generate a random integer between 0 and 25.
    Now add it to 65.

A translation of this is:

```
CharNum = INT(26*RND(1)) + 65
```

and here's a program that uses this to generate random four letter combinations:

```
'CH3\P14.BAS
' Random 4 letter `words'
' demonstrates RND for a range

DEFINT A-Z
RANDOMIZE TIMER
CLS

PRINT "This program generates random 4 letter combinations.  It"
PRINT "stops when you press any key."
PRINT "Press any key to start."
I$ = INPUT$(1)

DO
  WORD$ = ""
    FOR I = 1 TO 4
      CharNum = INT(26 * RND(1)) + 65          'A
      WORD$ = WORD$ + CHR$(CharNum)
    NEXT I
  PRINT WORD$
LOOP UNTIL INKEY$ <> ""                        'B

END
```

A) As explained in the outline this statement gives us a random integer in the right range—and the next statement turns it into a random uppercase letter. These two statements could have been combined into one:

```
Word$ = Word$ + CHR$(INT(26*RND(1)) + 65)
```

but this is clearly less readable than the above.

B) This is another way of saying LOOP WHILE INKEY$ = " ".

As another example of using this technique to get a range of values, suppose you wanted to write a typing reflex tester. You generate a random typewriter key and display it on the screen, then use INPUT$(1) or INKEY$ to grab the key that you pressed. To get a random typewriter ASCII code you need a number between 32 and 126. However, the spacebar doesn't show up too well on the screen, so the following program makes the range 33-126:

```
'CH3\P15.BAS
' a reflex tester

DEFINT I-N
RANDOMIZE TIMER

CLS
PRINT "I'm going to generate 10 different characters.  When the"
PRINT "character appears press that key (using the Shift key if"
PRINT "necessary."
PRINT
PRINT "The number correct will be listed at the end, followed by"
PRINT "the program's run time."
PRINT
PRINT "Press any key to start."

SLEEP

ANS$ = INKEY$
StartTime = TIMER
NumRight = 0

FOR I = 1 TO 10

  CLS
  N = INT(94 * RND(1)) + 33
  GuessKey$ = CHR$(N)
  LOCATE 12, 40: PRINT GuessKey$

  ANS$ = ""

    DO WHILE ANS$ = ""
       ANS$ = INKEY$
       IF ANS$ = GuessKey$ THEN NumRight = NumRight + 1
    LOOP

NEXT I

EndTime = TIMER

LOCATE 24, 1
PRINT "Your score was "; NumRight;
PRINT " - total runtime was "; EndTime - StartTime; "seconds."


END
```

I can't resist giving one last application of the RND function: abstract tiling. Recall that the CHR$(219) character is a solid block—and if a COLOR command has changed the foreground, then it's a colored block. This program generates a random foreground color and then paints the screen accordingly:

```
'CH3\P16.BAS
' Abstract designs

RANDOMIZE TIMER
CLS
PRINT "Press any key to see a random design.  When the screen is"
PRINT "filled, press any key to erase it and end."

FOR Rw = 1 TO 24
  FOR Col = 1 TO 80
   COLOR INT(16 * RND(1))
   PRINT CHR$(219);
  NEXT Col
NEXT Rw

SLEEP
CLS

END
```

Dramatic, isn't it?

Some of the other built-in functions

I'm not going to go over all the built-in functions. Now is a good time to take out the manual or browse through the on-line help and start skimming through it. On the other hand, where I can think of something useful that is not mentioned or not sufficiently stressed in the manual or in the on-line help, I will point it out in this section.

The screen functions

CSRLIN—This gives you the current line the cursor is on. Combine it with POS(1) function which gives the column the cursor is on to have a way to restore the cursor to its previous position, no matter how far it strayed:

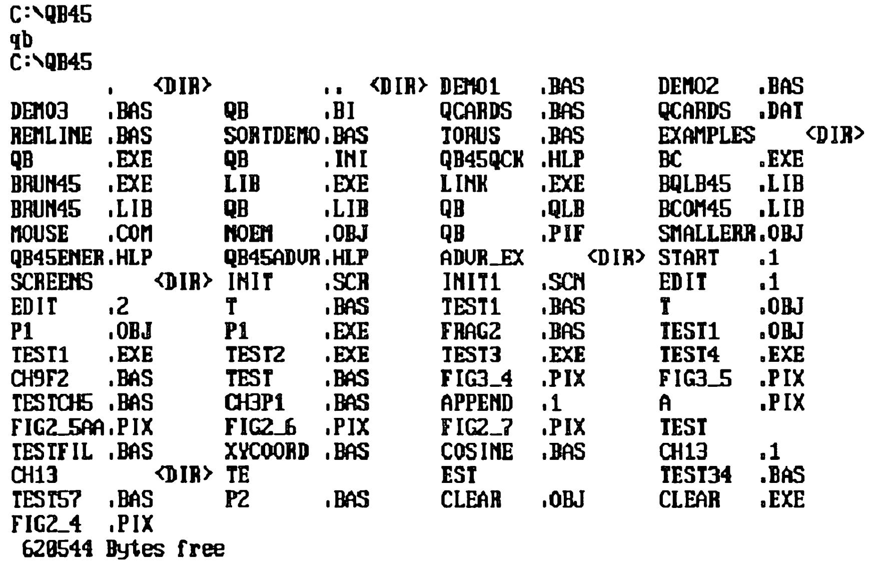
```
OldColumn = POS(1)
OldRow = CSRLIN
LOCATE OldRow,OldColumn   'back to where it belongs.
```

The SCREEN(row,column) function gives you the ASCII code of the character at location row, column. If that space is blank (for example, because of a CLS command), then the value of SCREEN is 32 (ASCII for a space).

An ultimately silly, but still one of my favorite applications of the SCREEN function is to—completely within QuickBASIC—find the amount of free space that's on the disk. The idea is the following. QuickBASIC has a command FILES that gives you a list of the files on the current directory—it works like a DIR/W command from DOS. To find the files on another directory, for example, on the directory C: \ QB \ Programs, say:

```
FILES "C:\QB\Programs"
```

You can also use standard DOS wild cards: FILE "A:*.BAS" would give all files on the A drive with the .BAS extension. Figure 3-1 is an example of what the FILES command gives. Notice that the number of bytes is given on the last line.



```
C:\QB45
qb
C:\QB45
        .    <DIR>         ..   <DIR> DEMO1    .BAS       DEMO2    .BAS
DEMO3    .BAS       QB       .BI       QCARDS   .BAS       QCARDS   .DAT
REMLINE  .BAS       SORTDEMO.BAS       TORUS    .BAS       EXAMPLES      <DIR>
QB       .EXE       QB       .INI      QB45QCK  .HLP       BC       .EXE
BRUN45   .EXE       LIB      .EXE      LINK     .EXE       BQLB45   .LIB
BRUN45   .LIB       QB       .LIB      QB       .QLB       BCOM45   .LIB
MOUSE    .COM       NOEM     .OBJ      QB       .PIF       SMALLERR.OBJ
QB45ENER.HLP        QB45ADVR.HLP       ADVR_EX       <DIR> START    .1
SCREENS      <DIR> INIT      .SCR      INIT1    .SCN       EDIT     .1
EDIT     .2         T        .BAS      TEST1    .BAS       T        .OBJ
P1       .OBJ       P1       .EXE      FRAG2    .BAS       TEST1    .OBJ
TEST1    .EXE       TEST2    .EXE      TEST3    .EXE       TEST4    .EXE
CH9F2    .BAS       TEST     .BAS      FIG3_4   .PIX       FIG3_5   .PIX
TESTCH5  .BAS       CH3P1    .BAS      APPEND   .1         A        .PIX
FIG2_5AA.PIX        FIG2_6   .PIX      FIG2_7   .PIX       TEST
TESTFIL  .BAS       XYCOORD  .BAS      COSINE   .BAS       CH13     .1
CH13         <DIR> TE                  EST                 TEST34   .BAS
TEST57   .BAS       P2       .BAS      CLEAR    .OBJ       CLEAR    .EXE
FIG2_4   .PIX
 620544 Bytes free
```

3-1 Using the files command.

Now imagine you clear the screen and issue the FILES command. To find out the number of free bytes all you have to do is:

Start out at the bottom line.
Check if the second entry is not blank.
  If it is, then this line contains the information you want.
  If not, move up a line and ask the question again.

Here is a program that translates this:

```
'CH3\P17.BAS
' Finding free disk space
' cute but inefficient

DEFINT A-Z
CONST SpaceCode = 32
Row = 25           'start from the last line
```

```
CLS

PRINT "Enter the drive identifier or pathname."
INPUT "For example A: for the A drive"; Drive$

CLS
FILES Drive$

TestChar = SCREEN(Row, 2)
DO WHILE TestChar = SpaceCode        'A
  Row = Row - 1
  TestChar = SCREEN(Row, 2)
LOOP

Col = 2
TotalBytes$ = ""

DO                                    'B
  Bytes = SCREEN(Row, Col)
  Bytes$ = CHR$(Bytes)
  TotalBytes$ = TotalBytes$ + Bytes$
  Col = Col + 1
LOOP UNTIL Bytes = SpaceCode

CLS
LOCATE 1, 1: PRINT "Total bytes free is"; VAL(TotalBytes$)

END
```

A) This loop looks at the first character code in the second column in each row—starting from the bottom row. The loop ends when it picks up a code different from 32 (ASCII space).
B) This loop reads the digits of free space. When a space shows up the loop is done.

To be honest this program is more a demonstration of the techniques that you've seen to this point than a good way to find out the amount of free space on a disk. Besides messing up the screen it takes far too long. The FILES command will be very useful though. I hope Microsoft soon makes "free disk space" into a built-in function. The best way to find out the amount of free disk space, is to use a DOS function call. It only takes a few lines, and you can find it in chapter 7.

Some numeric functions

If you don't do a lot of scientific work, then you'll be less likely to use this information. As it is I'm only describing a handful of QuickBASIC's numeric functions.

ABS( ) is the absolute value of whatever is inside the parentheses. All this function does is remove minus signs.

ABS(−1) = 1 = ABS(1)

One common use of it is in the form of ABS(B-A), when it gives the distance between the numbers A and B. For example, suppose A = 3 and B = 4, then:

```
ABS(A-B) = ABS(B-A) = 1
```

because 3 and 4 are one unit apart. As another example:

```
ABS(ASC(A$) - ASC(B$) )
```

gives the distance between the first two characters of the strings A$ and B$.

LOG( ) gives the natural logarithm of a number. To find the common log (log to base 10), use:

```
LOG10( )=LOG( )/LOG(10)
```

which gives the common logarithm of the value inside the parentheses. Another way to find the number of digits is to use (for number >1):

```
INT( LOG10(number) ) + 1
```

For example, LOG10(197) is between 2 and 3, because LOG10(100) is 2 and LOG10(1000) is 3.

Also, for those who need them, QuickBASIC has the built-in trigonometric functions SIN( ) (sine), COS( ) (cosine), and TAN( ) (tangent). The only problem is that QuickBASIC expects the angle inside the parentheses to be in radian measure. To convert from degrees to radians you need the value of π. The formula is:

$$\text{radians} = \text{degrees}*\pi/180$$

To get the value of π, the easiest method is to set up a variable PI# using the ATN (arctangent function), in the form of:

```
PI# = 4*ATN(1)
```

The arctangent of 1 is $\pi/4$. Sometimes you may want to set this value to be the value of a named constant.

You can also use the ATN function to get all the other inverse trigonometric functions. A list is given in the last appendix.

QuickBASIC has two built-in functions that are useful for dealing with integers and long integers. The first integer division is denoted by a backslash (\ ) (the same character as the DOS path separator). Some examples:

```
7\3    = 2    10\3    = 3
12\4   = 3    97\11   = 8
```

Integer division disregards the remainder after dividing, so it always ends up with an integer. To find the remainder you use the MOD function. Some examples:

```
7 MOD 3   = 1    10 MOD 3   = 1
12 MOD 4  = 0    97 MOD 11  = 9
```

For example, if you add two one digit numbers (like 9 and 7) then the carry is

(9+7) \ 10 and the ones digit is (9+7) MOD 10. Simple as this idea is, this is the key to the infinite precision addition program of the next chapter. Those who are familiar with the MOD function from mathematics are aware that MOD does not work for negative numbers: −1 MOD 5 is still −1, not 4.

Finally, as you saw in the first chapter, QuickBASIC doesn't tolerate intermediate overflow very well, operations that work with integers that, along the way, go out of bounds. The cure for this is the CDBL( ) (Convert DouBle) or CLNG functions mentioned earlier. This converts the numeric expression inside the parentheses to double precision (or a long integer)—overflow will no longer be a problem. However if A% = 32000 and B% = 32000 then you can't just say:

```
Answer = CDBL(A% + B%)
```

because the CDBL function only goes to work after the two numbers were added. Instead say:

```
Answer = CDBL(A%) + B%
```

which avoids the intermediate overflow. You can also convert numbers; in this case, CDBL(2345) works the same as saying 2345#. What goes on internally when you say CDBL is a bit tricky, but basically QuickBASIC converts its internal representation of the number to a new form. In particular this means that CDBL(X%) takes up more space than X% did.

If the number is small enough then you can convert to an integer (CINT), a long integer (CLNG), or a single precision number (CSNG). When you do this conversion the appropriate rounding takes place. Finally, let me emphasize that when you use CDBL you do not gain precision. Converting a single precision variable may allow you more room—or may give you more digits, but only the first six can be trusted. For example, try the following:

```
A  = 7.1
B# = 7.1
PRINT A
PRINT B#
PRINT CDBL(A)
```

What you'll see is:

```
7.099999904632568
7.1
7.099999904632568
```

The point is that when QuickBASIC stores a number like 7.1 as a single precision number, because of the way its internal arithmetic works, it usually uses an approximation. Only a double precision number would usually be accurate enough not to make any changes. And changing a number to double precision using CDBL doesn't magically make it more accurate.

Finally, I should mention that if you have the appropriate numeric co-processor for your computer, then many of these numeric functions will be done by the 87 chip. In fact, this chip has built into it ways of computing functions like SIN, EXP, or ATN—and they are fast. Otherwise, QuickBASIC imitates what the 87 chip would do in the main processor—and this can take 100 times as much time.

4
CHAPTER
User-defined functions

In the last chapter you saw how to use most of QuickBASIC's built-in functions. This chapter shows you how to create new functions—essentially, to add new commands to QuickBASIC. To do this you'll see the FUNCTION command as well as the older DEF FN (DEFine function) command.

Getting started

Start thinking about defining your own functions when you use a complicated expression more than once in a program. For example, suppose you need a random integer between 1 and 10. You could write:

```
INT(10*RND (1))+1
```

each time you needed it, although this might eventually grow tiresome. Suppose the same program needs random integers between 1 and 40, between 1 and 100, etc. The statements needed for these are so similar to the above statement, that you would hope that there is a way to automate the process, and have QuickBASIC do some of the work. What you do is add the statement:

```
DEF FNRdInt%(X) = INT(X*RND(1) + 1
```

to your program before you need any random integers. Now to print out a random integer between 1 and 10, say:

```
PRINT FNRdInt%(10)
```

(in computerese we say that you've called the function). The DEF is there to remind you of the word define, and the FN to remind you of the word function. The name of the function, in this case RdInt, must follow the same rules as for variables in QuickBASIC. That is why you can't have a variable whose name begins with FN—QB thinks you're calling a user-defined function. Just like for variables, you may have a type identifier at the end of the name of a function. In this case the % shows the RdInt% function has integer type. The type identifier determines the function's eventual value.

The key to this function's smooth operation is the X. It has a fancy name—it's called a *formal parameter*, but think of it as a placeholder. To call this random integer function you replace the formal parameter (the placeholder) by a numeric expression (variable, number, calculation). What QuickBASIC then does is replace all occurrences of the placeholder in the definition of the function by the value of the numeric expression. It does any calculations called for, so if:

```
A = 3 and B = 2
```

then:

```
N% =FNRdInt%(A*B + 37)
```

has the same effect as:

```
N%=FNRdInt%(43)
```

which in turn is the same thing as saying:

```
N%=INT(43*RND(1)) + 1
```

The value you sent the function is sometimes called the actual parameter. You shouldn't expect to get too large a random integer out of FNRdInt%, because FNRdInt%, like any integer expression, can't be larger than 32767.

The X used as a parameter or placeholder in the definition of the function has no independent existence. If you used an X as a variable somewhere earlier, say by setting X = 10, then this assignment never affects the value of the function.

FNRdInt% is passed the value of one piece of information. In general, the values QuickBASIC passes to a function are determined by a type identifier at the end of a placeholder, or by any DEFINT, (DEFLNG,DEFDBL, etc.) that happens to be in effect. In particular, I probably should have changed the definition of the function to be:

```
DEF FNRdInt%(X%) = INT(X%*RND(1)) + 1
```

Now the values you can send to the function—what eventually replaces the integer placeholder X%—must be within the range of QuickBASIC short integers: −32,768 to 32,767. The value of any numeric expression within this range that you send the function as a parameter will, of course, be rounded off.

The function FNRdInt% works with one piece of information—it's a function of one variable, or *argument*. You frequently want the value of a function to depend on more than one piece of information. For example, suppose you wanted a range of random integers between X and Y. Then we can modify FNRdInt% as follows:

```
DEF FNRdIntRange%(X,Y) = INT((Y - X + 1)*RND(1)) + X
```

This may seem a little tricky; if so, try to see what happens with numbers like 5 and 37. Multiplying RND(1) by Y − X + 1 (33) gives a range between 0 and 32 (= Y − X). Finally, add X to get the range wanted (5 − 37).

To get a range suitable for the ASCII typewriter codes use:

```
FNRdIntRange%(33,126)
```

When QuickBASIC processes this statement, it looks up the definition of the function, gives the placeholder X the value 33, and the placeholder Y the value 126. The result is that any time you call this function, a random integer between 33 (ASCII code for the !) and 126 (ASCII code for a ~) pops out. If you want to make sure that the function only uses integer values, rewrite it as:

```
DEF FNRdIntRange%(X%,Y%) = INT( (Y%-X% + 1) *RND(1)) + Y%
```

Now the placeholders can only have integer values. If you set:

```
Number% = IntRange%(2.7,39.2)
```

then 2.7 is rounded up to 3 and 39.2 is rounded down to 39, when QuickBASIC substitutes their values into the function definition. If you said:

```
Number% = IntRange%(3, 300000)
```

then you'd get an overflow error.

Since the name of a function, i.e., what follows the letters FN, follows the same rules as for the names of variables, choose meaningful function names; these will certainly make your program more readable, as well as easier to debug. The only thing to keep in mind is that, unless you give it an explicit type identifier, then the type of the function is determined by a type identifier or any DEFtype statements that may currently be in effect. One way to avoid any problems is to put the function definitions before any DEFtype statements. I think the best way, however, is to stick the appropriate type identifier at the end of every function name—overruling any DEFtype that may be in use.

If you are writing a program that needs one of the inverse trigonometric functions that are not built into QuickBASIC, make a user-defined function out of it. For example, to define the inverse sine (Arc Sine):

```
DEF FNArcSine#(Angle#)=ATN(Angle# - SQR(1 -Angle#*Angle#))
```

Just like the ATN function on which it is based, this function works with radians. This means you will have to do the conversion from degrees to radians, perhaps from another function, before calling this function.

Obviously you need to check the information you send a function before you call the function. With the Arc Sine:

```
IF(ABS(Value) < = 1 THEN
    PRINT FNArcSine (Value)
ELSE
    BEEP
    PRINT "Can't find an angle whose sine is greater than one!!!"
END IF
```

Check the values you send to functions before you call the function, otherwise you risk fatal error.

There are no restrictions on where you DEFine functions, other than you must DEFine it before you use it. A program is clearer if all the function definitions are grouped together.

You can define functions of as many as 16 variables; in computerese, having 16 formal parameters. For example, the volume of a box: FNVolOfBox(L,W,H) = L*W*H has three formal parameters.

Here's a program that uses a three variable function to solve quadratic equations. For the quadratic equation:

$$AX^2 + BX + C$$

the formula is:

```
Answers =(-B ± SQR(B*B -(4*A*C))/2*A
```

The A, B, and C are called the coefficients of the equation, and the numeric expression inside the square root is called the discriminant of the equation. If it's less than zero, then the quadratic has imaginary roots, i.e., the answers involve $\sqrt{-1}$="i". Here is a program for this:

```
'CH4\P1.BAS
' A quadratic equation solver
```

```
DEFDBL A-Z                                                    'A
CLS
Tplate$ ="+#########,.#######"

DEF FNDisc(A,B,C) = B*B - (4*A*C)                            'B

INPUT "Enter the three coefficients for the quadratic";A,B,C  'C

Discrim = SQR(ABS(FNDisc(A,B,C)))                             'D

IF FNDisc(A,B,C) < 0 THEN                                     'E
   PRINT "roots are imaginary"
   PRINT "First Root is  "; -B /(2*A)
   PRINT USING Tplate$ +"i";(-B +Discrim)/(2*A)
   PRINT "Second Root is "; -B /(2*A)
   PRINT USING Tplate$ + "i";(-B - Discrim)/(2*A)
 ELSE
   PRINT "roots are real"
   PRINT "First Root is  ";
   PRINT USING Tplate$;(-B + Discrim)/(2*A)
   PRINT "Second Root is ";
   PRINT USING Tplate$;(-B - Discrim)/(2*A)

END IF

END
```

A) This DEFine type command affects all the variables and the function that follows.
B) As mentioned above, the discriminant is a function of the three coefficients. Here they are all double precision variables because of A.
C) Although it takes getting used to, the A, B, C used here have no relation to the A,B,C used as placeholders in the function's definition.
D) This sets up a variable whose value is closely related to the discriminant. It's here to make the program more readable.
E) Now the values of the variables A,B,C are sent to the function. Notice the PRINT USING command (with a string), to print the "i" flush with the square root. I'm allowing 9 digits before the decimal point and 6 digits after the decimal point. I decided to print a plus or a minus sign as well as using commas, if needed.

You can work with numbers and get strings, or use numbers and strings to get numbers; all possible combinations are allowed. However, I won't bother giving examples because I want to immediately turn to the next section.

First steps in multi-line functions

Older interpreted BASIC's were limited to the kind of functions described in the previous section. And, as you can well imagine, there's a limit to what you can do in a single line. QuickBASIC on the other hand, allows you to take as many steps as you want in defining a function. For example, to find the largest of two numbers, use:

```
DEF FNLarge(X,Y)
    IF X < Y THEN FNLarge = Y ELSE FNLarge = X
END DEF
```

As the above example indicates, you begin defining a multi-line function by using the key word DEF, followed by the name of the function and the formal parameters, up to 16 are allowed, that it's going to use. However, unlike the situation with the simple one line functions from the last section, you immediately hit the Enter key and begin the statements that will define the function. Next you make an assignment that will determine the value of the function. In the above example, the assignment that is made depends on which of the two numbers is larger. As this example indicates, when you make the assignment that defines the function, you don't say:

```
    FNLarge(X,Y) = ...
```

you just use the function's full name:

```
    FNLarge = ...
```

The statement END DEF marks the end of the definition and, just as with loops, the statements between the DEF and the END DEF are called the body of the function. If you make more than one assignment to the function within the body of the loop, only the last one counts.

A multi-line function can be useful if you need many PRINT USING templates. A function that sets up the number of decimal digits for any number might look like:

```
DEF FNTemplate$(Number,DecDigits)
    IF Number > 0 AND Number < 1 THEN
        FNTemplate$ = "#." + STRING$(DecDigits,"#")
    ELSE
        Digits = INT(FNLOG10(ABS (Number))) + 1
        FNTemplate$=STRING$(Digits,"#") + "." + STRING$(DecDigits,"#")
    END IF
```

where FNLOG10 is the function that you have previously defined following the outline from the last chapter. By putting the ABSolute value inside the parentheses you've taken care of negative numbers as well.

Suppose you wanted to write a function that would allow you to chop out any substring. You saw how to do this in the last chapter: use INSTR to find out where the string was and then use RIGHT$, LEFT$, and MID$ to do the cutting. Here's one way to try to write the function:

```
DEF FNCutStr$(Big$,Small$)
    IF INSTR(Big$,Small$) = 0 THEN
        FNCutStr$ = Big$
    ELSE
        FNCutStr$ = LEFT$(Big$, INSTR(Big$,Small$) - 1) +
        MID$(Big$,INSTR(Big$,Small$) + LEN(Small$) + 1)
    END IF
END DEF
```

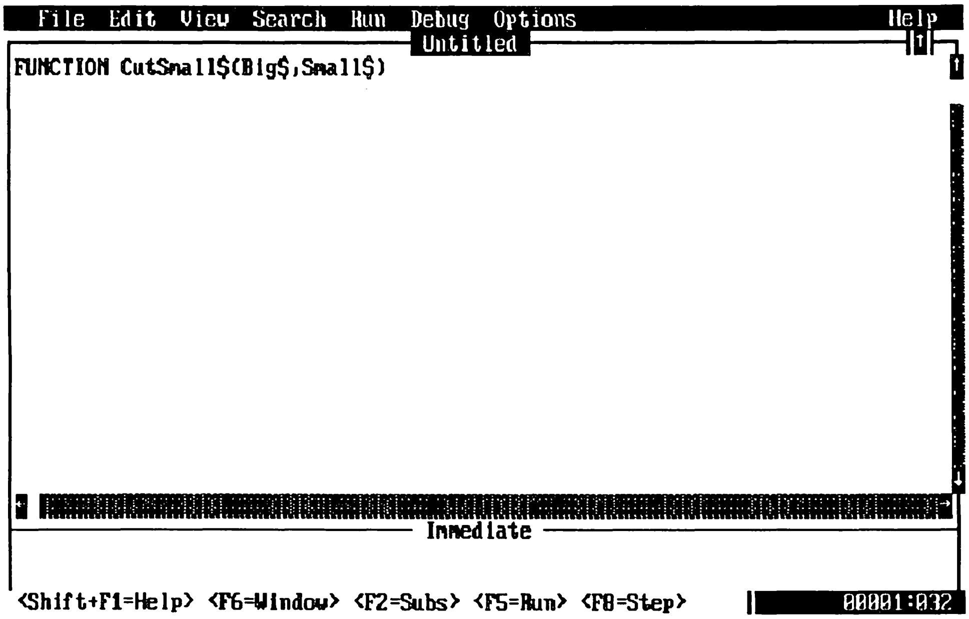
To actually enter this into QuickBASIC, you'd have to go past the limits on a single line. It would be necessary to scroll horizontally. This puts this definition at the limits of readability—I think it has gone past it. To make it more readable and be in a position to fully exploit the power of user-defined functions, you need to use temporary variables within them. To do this properly you have to switch from the older DEF FN structure to a newer one called *FUNCTION PROCEDURES*.

The DEF FN construct is for the most part still in QuickBASIC for compatibility with interpreted BASIC. FUNCTION procedures can do everything that DEF FN can do, and more. Using this new construct, QuickBASIC allows you to set up variables, called local variables, that exist only within a function. Each time the function is called they are re-initialized—they'll start over from scratch. Here's a much more readable version of the previous function, but hold off entering it for a moment:

```
    FUNCTION CutSmall$(Big$,Small$)
        Place = INSTR(Big$,Small$)
        Length = LEN(Small$)
        IF Place = 0 THEN
            CutSmall$ = Big$
        ELSE
            CutSmall$ = LEFT$(Big$,Place - 1) + MID$(Big$,Place + Length)
        END IF
END FUNCTION
```

The local variables named Place and Length here have no effect on any other variable named "Place" that might occur elsewhere within the program. I said that you shouldn't try it yet. This is because something strange happens once you enter this. Figure 4-1 shows what your screen looks like after you type the first line but before you hit Enter. Now hit Enter. Notice the screen has changed dramatically, and now looks like Fig. 4-2. What has happened is that QuickBASIC has opened a new window to edit your FUNCTION. This is indicated by the name. It's:

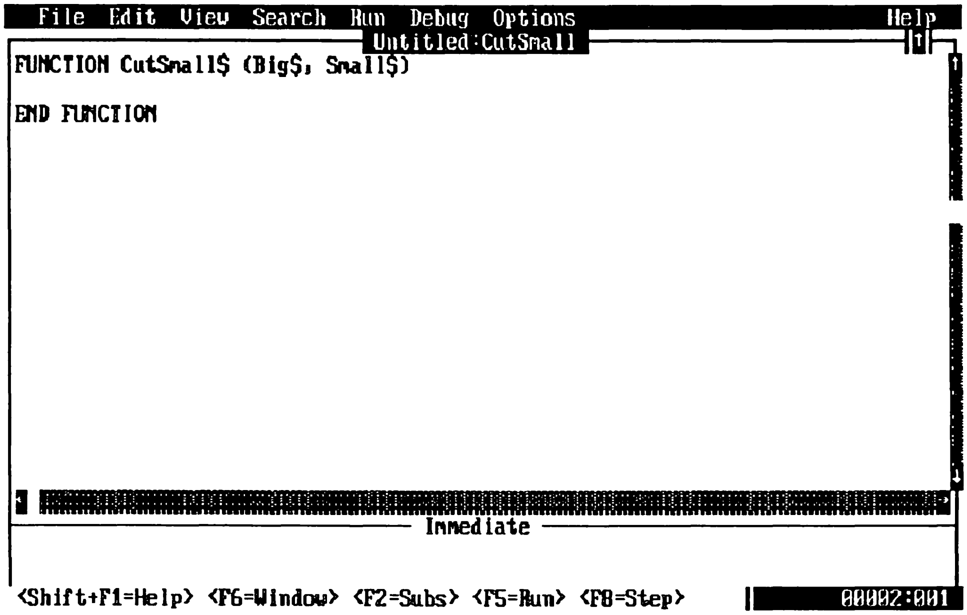
Untitled:CutSmall



```
FUNCTION CutSmall$(Big$,Small$)
```

```
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>      00001:032
```

4-1 The screen before you hit Enter.



```
FUNCTION CutSmall$ (Big$, Small$)

END FUNCTION
```

```
<Shift+F1=Help> <F6=Window> <F2=Subs> <F5=Run> <F8=Step>      00002:001
```

4-2 The screen afterward.

The colon separates the name of the subordinate function from the name of the program, or main module, as it's usually called.

Anyway, type the function exactly as I've written it. O.K., you're done typing and want to test this function. Hit SHIFT+F5 just as before, and the program runs, but nothing has been fed to the function, so nothing happens. You need to return to the main module to use this FUNCTION procedure. To do that hit SHIFT+F2. Notice that QuickBASIC has closed the previous window and returned you to the main module, sometimes called the module level code, i.e., square 1. SHIFT+F2 cycles through the various subordinate functions that you've written, and the main module. Now enter the line:

```
PRINT CutSmall("This is not a test", "not ")
```

and run the program.

Once you've finished a FUNCTION, you can also use the Immediate window to test it. For example, you could have moved to the Immediate window and typed the line above.

Assuming you haven't made any typos, then this program should run. So suppose we try to save it—give a name and then open the FILE menu to do the save. Notice that after the save QuickBASIC has added a line:

```
DECLARE FUNCTION CutSmall$ (Big$,Small$)
```

at the top of the program. DECLARE statements are used by the compiler for bookkeeping. In this case the compiler is keeping track that you are using a function of two string variables. QuickBASIC automatically adds the appropriate DECLARE statement whenever you save the program.

While developing your program within the QuickBASIC environment, you rarely need a DECLARE statement. However, it's good programming practice to include it. This is because DECLARE statements are essential in creating certain kinds of standalone programs. More on that later.

As another example, suppose you wanted to write a function which counted the number of times a character appeared in a string. This would be a function of two string variables, and it should return an integer. It's called CharCount%:

```
FUNCTION CharCount%(X$,Y$)

'This function counts the number of times
'the character Y$ is inside the string X$
'If Y$ is not a character or Y$ does not occur in
'X$ then this function returns zero
' Three local variables: I%, Count% and LenString
Count% = 0
LenString% = LEN(X$)

FOR I% = 1 TO LenString%
   IF MID$(X$,I%,1) = Y$ THEN Count% = Count% + 1
NEXT I%

CharCount% = Count%

END FUNCTION
```

First off, notice the extended REMark section. In defining a complicated function, you're best off explaining what's supposed to happen. Explain what kind of information the function expects to deal with, what local variables it uses, and what it's supposed to send out. If you know what the function expects, then you're more likely to check what you send it, before the program blows up in your face. Most of the example programs in this book have, to this point, been sparsely commented—mostly because the surrounding text (hopefully) explained it. However, when you're hired to write a program, this is the way you'd be expected to comment it. In fact you should probably explain what the local variables were doing as well.

This function uses three local integer variables, Count%, I% and Len String%. It's good programming practice to initialize the local variables before going on to the main business of the FUNCTION. Being local variables they have no connection with any variables that might share the same name elsewhere in the program. The advantages this gives over the older DEF FN can't be stressed enough. A complicated string handling program might have 17 different functions with 17 different local variables named LenString% or Count%, and you don't want their values contaminating each other.

Next, notice that I initialized the Count% variable to be 0. I did this for the same reason that I'd initialize a variable in the main part of a program; relying on default values is sloppy, and dangerous. LenString% doesn't change in the loop that follows, so, since it is a loop invariant, I calculate it once, before the loop starts, in order to speed the function up.

The FOR-NEXT loop runs through the string, character by character, checking if it has a match—and adding one to the Count% if so. Finally, the value of the local variable Count% is the information this function will return. In this case I end the body of the function with the assignment that defines the function.

Using a variable such as Count% to accumulate information as the function works is quite common—when you're done you use the accumulator in the final assignment that determines the value of the function.

As another example, return to the "find the next word" program from the last chapter. Suppose you want to modify this program so that it gives the next word, no matter what separator you use. Using a user-defined function makes this almost trivial. All you have to do is replace the statement:

```
AfterSpace = INSTR(BeforeSpace+1,Phrase$,Separate$)
```

by a function that finds the next separator. Before I show you an outline for this function, think about the information this function needs to manipulate. Is it clear that it will work much like INSTR, except that the separators will be built into the function? Once you convince yourself of this, then understanding the following outline for the function shouldn't be a problem.

The function works with a string and a position number.

It should search starting at the position after "position number," and
Look character by character until it finds a separator.
If successful the function should return its position number.
If not it should return zero.
Here is a function definition that follows this outline:

```
FUNCTION FindSeparator%(Phrase$,Position%)

  AfterSpace% = Position% + 1                       'A
  Answer% = 0                                       'B
  LenString% = LEN(Phrase$)

  DO UNTIL Answer%< >0 OR AfterSpace% > LenString%  'C

     NxtChar$ = MID$(Phrase$,AfterSpace%,1)

     SELECT CASE NxtChar$

      CASE CHR$(32),"!","?"                   'D
        Answer% = AfterSpace%
      CASE  ".",";",":"
        Answer% = AfterSpace%
      CASE ","
         Answer% = AfterSpace%
      CASE ELSE                                  'E
        AfterSpace% + AfterSpace% + 1
     END SELECT

  LOOP

  FindSeparator% = Answer%                    'F
END FUNCTION
```

A) It starts looking from one position further along in the string.
B) This local variable plays a key role in what follows. As it moves through the loop while this variable accumulates information, either by staying equal to zero, in which case no separator was found, or made positive, i.e., value would be where the separator is.
C) This kind of loop has to stop if either the accumulator tells it the hunt was successful or there's no place else to look, because the loops finished searching all the characters in the string. Notice that instead of using a eureka loop you could have used multiple EXIT LOOP commands.
D) All the tests done in the next three cases could have been put on one line, but the program would have looked awful. If you can think of a separator I left out, add it on!
E) If the character wasn't a separator the loop moves on to the next character.
F) The value of the accumulator variable Answer% will be zero if no separator was found, otherwise it will be the position of the separator.

Here is a revised find word program (notice the DECLARE statement at the top):

```
DECLARE statement at the top):
DECLARE FUNCTION FindSeparator% (Phrase$, Position%)

' CH4\P2.BAS
' This program uses the function
' FNFindSeparator to parse a phrase
' by searching for the separators
' . , : ? ; ! and the space
' it finds all words contained in the
' phrase

CLS

DO
 PRINT "Please enter the phrase to parse."
 LINE INPUT "?"; Phrase$
 LenPhrase% = LEN(Phrase$)
LOOP UNTIL LenPhrase% > 0

BeforeSpace% = 0
AfterSpace% = FindSeparator%(Phrase$, BeforeSpace%)

DO UNTIL AfterSpace% = 0

  SizeOFWord% = AfterSpace% - BeforeSpace% - 1
  NextWord$ = MID$(Phrase$, BeforeSpace% + 1, SizeOFWord%)
  IF SizeOFWord% > 0 THEN PRINT NextWord$                     'A
  BeforeSpace% = AfterSpace%
  AfterSpace% = FindSeparator(Phrase$, BeforeSpace%)

LOOP

      PRINT MID$(Phrase$, BeforeSpace% + 1)                   'B

END

FUNCTION FindSeparator% (Phrase$, Position%)

  AfterSpace% = Position% + 1
  Answer% = 0
  LenString% = LEN(Phrase$)

  DO UNTIL Answer% <> 0 OR AfterSpace% > LenString%

     NxtChar$ = MID$(Phrase$, AfterSpace%, 1)

     SELECT CASE NxtChar$

      CASE CHR$(32), "!", "?"
        Answer% = AfterSpace%
      CASE ".", ";", ":"
```

```
        Answer% = AfterSpace%
      CASE ","
        Answer% = AfterSpace%
      CASE ELSE
        AfterSpace% = AfterSpace% + 1
    END SELECT

  LOOP

  FindSeparator% = Answer%

END FUNCTION
```

A) As mentioned before, this takes care of two spaces or two separators occurring together.

Finally, let me end this section by pointing out that many of the example programs that you've seen to this point were really functions in disguise. A good example was the super input program from the last chapter (CH3 \ P7.BAS) that accepted numbers, disregarding commas and decimal points.

Nothing prevents a function from calling another function, and you'll find yourself doing this frequently. A good exercise would be to change the find all words program to be a "find next word" function; it will still use the find separator function. However, while QuickBASIC allows you to call as many functions as you want from within a given function, you can't nest function definitions. Only one FUNCTION may be defined at any one time.

Functions however may call themselves. This is called recursion, and is so important (and fascinating) that the topic deserves a chapter of its own (chapter 7).

Shared and static variables

The purpose of using local variables is to avoid inadvertent side effects. A side effect is when something you do in the function affects the rest of the program. For example, if you place a CLS (clear screen) command in a FUNCTION, then every time the function is called the screen will clear; clearly a side effect.

There's certainly nothing wrong with a controlled side effect. But the key word is controlled. You must know exactly when they're going to happen and what the fallout is for the rest of the program.

One way to avoid problems is to use what are called shared variables, also called global variables. These variables are another way to communicate information between a program and a function besides the parameters.

When you declare a variable as SHARED it is visible to both the function and the main module. Any changes you make to this variable will persist in the main module. For example, a program might need the value of π in 20 different functions, so it's a perfect candidate for a shared variable:

```
FUNCTION AreaOfCirc(R)
    SHARED Pi#
    AreaOfCirc = Pi#*R*R
END FUNCTION
```

Which assumes Pi# has been defined in the main module.

As another example, consider a program that will massage the same phrase in 100 different ways—using 100 different functions. It's quite likely that the length of the phrase will pop up repeatedly, so rather than waste time re-computing it each time, make it a SHARED variable.

Once a variable is SHARED then any changes you make to it persist. For example, consider the following two test programs:

```
'CH4/P3.BAS
' A demonstration of local variables

DECLARE FUNCTION CountEses%(X$)

DEFINT A-Z
Count = 0
PRINT Count
Phrase$ = " Mississippi "
LenPhrase = LEN(Phrase$)

PRINT CountEses(Phrase$)
PRINT Count

END

FUNCTION CountEses%(X$)

Shared LenPhrase

FOR I = 1 TO LenPhrase
  IF MID$(X$,I,1) = "s" THEN Count = Count + 1
NEXT I

CountEses = Count

END FUNCTION
```

You'll see:

```
0
4
0
```

Why does this program do this? The first PRINT statement gives 0, because that's the current value of Count%. Next, it prints the number of s's in Mississippi—4. Next the value of Count% is displayed again. It's still 0 because we made Count%

a local variable in the FUNCTION. The variable Count% in the main module has nothing to do with the one in the FUNCTION definition. Compare this to:

```
'CH4/P4.BAS
' A demonstration of shared variables

DECLARE FUNCTION CountEses%(X$)

DEFINT A-Z

CLS
Count = 0
PRINT Count
Phrase$ = " Mississippi "
LenPhrase = LEN(Phrase$)

PRINT CountEses(Phrase$)
PRINT Count

END

Function CountEses(X$)

Shared LenPhrase, Count

FOR I = 1 TO LenPhrase
  IF MID$(X$,I,1) = "s" THEN Count = Count + 1
NEXT I

CountEses = Count

END FUNCTION
```

where you make the Count a shared variable. Now calling the function changes the value of Count—it's now 4. So you'll see:

```
0
4
4
```

Note that if you enter this program, the Smart Editor would insert another DEFINT A-Z as the first line of the window where the function is defined. This is because QuickBASIC uses whatever DEFiners are in effect at a given point for functions created at that point. And, to emphasize this, the editor inserts the appropriate DEFiner as the first line, and makes the DECLARE statement, that it automatically generates, reflect this.

At this point our functions in QuickBASIC are manipulating the values of variables; they massage the actual parameters, they don't change them. In computerese this is called passing by value. In the next chapter you'll see how to change the value of parameters, called passing by reference.

In practice you should rarely have to change the value of a SHARED variable. In some sense a function should do one thing: it should manipulate values and perhaps make one new one. Changing a shared variable is best done by means of an assignment statement within the main body of the program. Your programs will be much easier to debug if you use shared variables only to communicate information to functions.

At this point don't try to assign values to the formal parameters. Use them strictly as placeholders. We'll talk more about functions like:

```
FUNCTION INCR (X)
    X = X + 1
END FUNCTION
```

in the next chapter.

There's one final type of variable that you'll occasionally use in a function: the STATIC variable. This variable is local to the function, so it doesn't affect the rest of the program. However, its value is not reset each time the function is called. A function using only STATIC variables will work a little faster, so QuickBASIC allows you to make all variables in a FUNCTION into STATIC variables. This is done by placing the key word STATIC on the end of the same line as the key word FUNCTION, as follows:

```
FUNCTION . . . . . . . . STATIC
```

However I rarely do this as I regard it as a better programming practice to explicitly say which variables are STATIC and which are SHARED. All the other variables within a FUNCTION are, of course, LOCAL but I also like to point this out in a remark statement.

In fact, I most commonly use STATIC variables as a debugging tool:

```
FUNCTION TestFunction
    STATIC TimesCalled
    .
    .
    .
    TimesCalled = TimesCalled+ 1
    PRINT "This function has been used"; Timescalled
END FUNCTION
```

Making the variable TimesCalled a STATIC variable means that it will keep track of the number of times I use the function, printing out the current value each time the function is called, or put it in the watch window. If TimesCalled was a LOCAL variable then the value would be re-set back to zero each time the function is called, and I lose the information it contains. Knowing how many times a function was called is often the key to diagnosing what's wrong with a program.

Finally, let me end this section by pointing out that you don't have to give every function a value. Sometimes you are forced to EXIT it prematurely:

```
FUNCTION Bailout(X)
    IF X < 0 THEN
     EXIT FUNCTION
    ELSE
    .
    .
    .
    END IF
    .
    .
    .
END FUNCTION
```

This function bails out if a negative value is sent to it. Now calling the function with a negative value gives the function the default value of any numeric variable—0. A string function that you bail out of would return the null string, "", as its value.

I rarely use the EXIT FUNCTION. And, I certainly don't use it as the example given above. I check out the information I want to send to a function before I call the function, possibly using another function. I use the EXIT FUNCTION much like the EXIT DO, if it makes the program clearer. For example, you can re-write the find separator function using an EXIT FUNCTION, if you thought it made the program clearer.

Some longer example programs

In this section I want to develop three example programs that showcase the techniques of this chapter, and the last. They are a bit more complicated than the programs that you've seen to this point.

Dawkin's evolutionary demonstration program

Some people argue against evolution by saying it is as likely as a team of monkeys sitting down at a typewriter, randomly pecking keys and the text of Shakespeare's Hamlet popping out. How likely is this? Well, just to get the title "Hamlet by William Shakespeare" requires about 100^29 tries, because there are about 100 keys on a typewriter and 29 characters in the title. Or, assuming a monkey can type 500 characters a minute, it would take around 40 E45 years. (100^29/500 gives the number of minutes, and divide by 60*24*365 to get years.) Since the maximum age scientists estimate the universe has been around is only 25,000,000,000,000 years, this argument would certainly cause evolutionists problems, if it had any merit.

But as Richard Dawkins points out in his superb book *The Blind Watchmaker* (Norton 1986), from which I borrowed the ideas used in this discussion, and which I can't recommend highly enough, this argument confuses what he calls "single step change" with "cumulative change." In single step change the changes are truly random. There is no connection between the previous generation and the next generation. In cumulative change the results of the previous generation are fed to some sort of sieve which selects only those that have some desirable property to persist.

Here (following Dawkins) is an outline for how the two types of change might be programmed:

```
Single Step Change
    Start with a random string of the right length
    Change one character randomly to obtain the child
    Repeat the previous step (a few zillion times)
    Until you get the phrase "Hamlet by William Shakespeare".
Cumulative Change
    Start with a random string of the right length
    Change one character randomly in the parent string to obtain the child
    Sieve out the parent or child depending on some survival rule
    Use the one that survived as the new parent and repeat the previous step
    Until you get the phrase "Hamlet by William Shakespeare"
```

Since the program that follows is just a demonstration of the difference between the two types of change, it uses a nonevolutionary, "teleogical" (goal directed) survival rule: The phrase that survives will be the one closest to the target phrase: "Hamlet by William Shakespeare".
Here is the program:

```
DECLARE FUNCTION DistToTarget% (A$, B$)
DECLARE FUNCTION Mutate$ (A$)
DECLARE FUNCTION RndInt% (A%, B%)

' CH4\P5.BAS
' Dawkins' cumulative selection demo

CLS
DEFINT A-Z
RANDOMIZE TIMER
GenNumber = 0
Target$ = "Hamlet by William Shakespeare"
Parent$ = STRING$(29, "*")                         'A

LenTarget = LEN(Target$)

DO UNTIL Parent$ = Target$
```

```
  ParentDist = DistToTarget(Target$, Parent$)     'B
  Child$ = Mutate$(Parent$)                        'C
  ChildDist = DistToTarget(Target$, Child$)

  IF ChildDist <= ParentDist THEN Parent$ = Child$     'D
  GenNumber = GenNumber + 1
  IF GenNumber MOD 100 = 0 THEN                    'E
    PRINT GenNumber, Parent$, ParentDist
  END IF

 LOOP

 PRINT "It took"; GenNumber; " generations to finish"
END

FUNCTION DistToTarget (A$, B$)

  ' has LOCAL variables CharDistance,TotalDistance,I

  SHARED LenTarget

  TotalDistance = 0

  FOR I = 1 TO LenTarget
    CharDistance = ASC(MID$(A$, I, 1)) - ASC(MID$(B$, I, 1))
    TotalDistance = TotalDistance + ABS(CharDistance)
  NEXT I

  DistToTarget = TotalDistance

END FUNCTION

FUNCTION Mutate$ (A$)

 ' has LOCAL variables RandomPos,RandomChar$,Temp$

SHARED LenTarget


Temp$ = A$
RandomPos = RndInt(1, LenTarget)
RandomChar$ = CHR$(RndInt(32, 126))

MID$(Temp$, RandomPos) = RandomChar$

Mutate$ = Temp$

END FUNCTION

FUNCTION RndInt (A, B)
    RndInt = INT((B - A + 1) * RND(1)) + A
END FUNCTION
```

A) This is a convenient way to start off; a string of length equal to the target phrase, consisting only of *.

B) This function will calculate the distance between two words, for instance X$ and Y$. What it will do is, character by character, add up the absolute values of:

ASC(MID$(X$,I,1,)) – ASC(MID$(Y$,I,1)

As I mentioned earlier in the previous chapter, this is a common use of the ABS function.

C) This "mutates" a single character by randomly changing one character, using the MID$ command, since the size doesn't change.
D) This is the survival rule. The phrase that's closest to the target survives.
E) This keeps track of what's happening every 100 generations. A number is 0 MOD another number, only when it's perfectly divisible by that number. If you want to see every generation, just use a PRINT statement here.
F) LenTarget is a perfect candidate for a shared or global variable.

If you run this program you'll notice that the parent quickly mutates close to the target, then it takes a much longer time to actually reach the target. The reason is simple: once we're close to the goal then it's more likely that the mutations are harmful.

A Pig Latin generator

For the second programming example I want to give a "pig latin converter." Pig latin is a well known variant on English that children use: ancay ouyay understandway isthay?
The rules are simple:

1. All one letter words stay the same.
2. Words beginning with vowels get the suffix "way".
3. Words beginning with a string of consonants have the consonants shifted to the end and the suffix "ay" added.
4. Any "q" moved because of rule 4, carries its "u" with it.
5. Y is a consonant.

An outline for a "Pig Latin" convertor could be:

```
While there are still words,
    "Pig latinize" the next word.
    To "Pig latinize" a word follow the rules given above.
```

Here is a program to do this:

```
'CH4/P6.BAS
' A pig latin generator
' This program translates a phrase into
' pig latin.  It modifies the `find word'
' program by adding a `latinize' function       (A)

DECLARE FUNCTION FindSeparator% (Phrase$, Position%)
DECLARE FUNCTION Latinfy$ (A$)
DECLARE FUNCTION ShiftCons$ (A$)

DEFINT A-Z
CONST True = -1
CLS

DO
 PRINT "Please enter the phrase to parse."
 LINE INPUT "?"; Phrase$
 LenPhrase = LEN(Phrase$)
LOOP UNTIL LenPhrase > 0

BeforeSpace = 0
AfterSpace = FindSeparator%(Phrase$, BeforeSpace)

DO UNTIL AfterSpace = 0

  SizeOFWord = AfterSpace - BeforeSpace - 1
  NextWord$ = MID$(Phrase$, BeforeSpace + 1, SizeOFWord)
  IF SizeOFWord > 0 THEN
    PigWord$ = Latinfy$(NextWord$)                        'B
    PRINT PigWord$ + SPACE$(1);
  END IF

  BeforeSpace = AfterSpace
  AfterSpace = FindSeparator(Phrase$, BeforeSpace)

 LOOP
      FinalWord$ = MID$(Phrase$, BeforeSpace + 1)
      PigWord$ = Latinfy(FinalWord$)
      PRINT PigWord$ + SPACE$(1);
END

FUNCTION FindSeparator% (Phrase$, Position%)

  ' LOCAL variables are AfterSpace%, Answer%,LenString%,NxtChar$

  AfterSpace% = Position% + 1
  Answer% = 0
  LenString% = LEN(Phrase$)

  DO UNTIL Answer% <> 0 OR AfterSpace% > LenString%

     NxtChar$ = MID$(Phrase$, AfterSpace%, 1)

     SELECT CASE NxtChar$

      CASE CHR$(32), "!", "?"
        Answer% = AfterSpace%
      CASE ".", ";", ":"
        Answer% = AfterSpace%
     CASE ","
        Answer% = AfterSpace%
     CASE ELSE
       AfterSpace = AfterSpace% + 1
      END SELECT

  LOOP

FindSeparator% = Answer%

END FUNCTION

FUNCTION Latinfy$ (A$)
  IF LEN(A$) = 1 THEN                                        'C
     Latinfy$ = A$
  ELSE
     FirstChar$ = UCASE$(LEFT$(A$, 1))

     SELECT CASE FirstChar$

       CASE "A", "E", "I", "O", "U"
         Latinfy$ = A$ + "way"
       CASE IS < "A"                                         'D
         Latinfy$ = A$
       CASE IS > "Z"
         Latinfy$ = A$
       CASE ELSE
         Latinfy$ = ShiftCons$(A$) + "ay"              'E

       END SELECT

   END IF

END FUNCTION


FUNCTION ShiftCons$ (A$)

  ' LOCAL variables Count,NextChar$,Done

  Count = 1
  Done = 0

  DO
   NextChar$ = UCASE$(MID$(A$, Count, 1))

   SELECT CASE NextChar$

     CASE "A", "E", "I", "O", "U"
      Done = True                                        'F
     CASE "Q"                                            'G
       Count = Count + 2
      CASE ELSE
        Count = Count + 1
   END SELECT
  LOOP UNTIL Done

  ShiftCons$ = MID$(A$, Count) + LEFT$(A$, Count - 1) 'H

END FUNCTION
```

A) This program is a good demonstration of how longer programs can be built up from building blocks; I'll have more to say about this in the next chapter.
B) As I mentioned above, you have to change the "find next word program" to one which, instead of printing the "next word," prints the converted form. I decided to strip out all punctuation and spaces along the way. I'll leave it to you to change the program so that the converted phrase retains the original punctuation.
C) This translates rule 1: one letter words remain the same.
D) Rule two is, words with a leading vowel add "way." The next case is to make sure numbers and other special characters are not transformed.
E) This calls the most complicated function, the one that shifts the consonants to the end.
F) I use a flag to detect when, moving letter by letter, it finally hits a vowel.
G) By adding two to the "count" it carries the "u" along with the "q."
H) By starting with count equal to one and INCrementing the count every time a consonant shows up, LEFT$(A$,Count−1) must, when the loop ends, contain the leading consonants.

Infinite precision arithmetic

For the final example program in this chapter I want to write an infinite precision "adder." As you know, the limits on even long integers prevent you from adding really large numbers accurately. Sometimes however you'll want to add or multiply hundreds of digits at a time. The easiest way to do this in QuickBASIC is to first think of the large number as a string of digits. Now do what you learned in elementary school:

Start from the right-most digit of each number.
Add the digits.
Compute the answer and the carry.
Add the next two digits, and the carry, if any.
Compute the answer and the carry.
Continue until there are no more digits.

Here is a function that takes two strings, consisting only of digits, and gives back a string that has the digits of the answer:

```
DEFINT A-Z
FUNCTION InfinAdd$ (A$, B$)
```

```
' This function takes two strings consisting only
' of digits with no leading or trailing spaces
' and returns a string which consist of the digits
' in the sum

' LOCAL variables are: LenA,DigInA,CurDigA LenB,DigInB,CurDigB
' C$, CurDigC, dig, Answer, NextCarry, PreviousCarry

LenA = LEN(A$)
LenB = LEN(B$)
DigInA = LEN(A$) - 1                         'A
DigInB = LEN(B$) - 1

Dig = 0

DO UNTIL Dig > DigInA AND Dig > DigInB

  IF Dig <= DigInA AND Dig <= DigInB THEN        'B

     CurDigA = VAL(MID$(A$, LenA - Dig, 1))
     CurDigB = VAL(MID$(B$, LenB - Dig, 1))
  ELSEIF Dig <= DigInA AND Dig > DigInB THEN
     CurDigA = VAL(MID$(A$, LenA - Dig, 1))
     CurDigB = 0
  ELSEIF Dig > DigInA AND Dig <= DigInB THEN
    CurDigA = 0
    CurDigB = VAL(MID$(B$, LenB - Dig, 1))
  END IF

   ANS = CurDigA + CurDigB + PreviousCarry       'C
   NextCarry = ANS \ 10
   DigC = ANS MOD 10
   DigC$ = STR$(DigC): DigC$ = RIGHT$(DigC$, 1)    'D
   C$ = DigC$ + C$

   PreviousCarry = NextCarry
   Dig = Dig + 1

LOOP
IF NextCarry <> 0 THEN
  Last$ = STR$(NextCarry)
  C$ = RIGHT$(Last$, 1) + C$
END IF

InfinAdd$ = C$

END FUNCTION
```

A) The number of digits in a number is one less than the length.
B) Since I chose not to pad the smaller number, so that both numbers have the same number of digits, by adding leading zeroes, I needed this somewhat complicated block IF-THEN to take care of the three possibilities.
C) At any given moment I need to keep track of the previous carry and the

future carry. This line does exactly what you learned to do in elementary school.

D) Don't forget the extra space that the STR$ command sticks on at the front of a converted number.

Now to turn this into a suitable "infinite adder" you only need to add a function that accepts a string consisting of only digits, and you've already seen how to do this (CH2\P5.BAS). Since QuickBASIC allows strings to have as many as 32767 characters, you can add two rather large numbers together. This program is reasonably fast: it adds two 10,000 digit numbers in about two minutes on a PC AT.

A program that does infinite precision multiplication is similar and at least, if done naively, somewhat easier. You can imitate the elementary school method, calling the infinite adder function as needed.

Actually there's a slightly faster way to do infinite precision multiplication, rather than blindly following the method you learned in elementary school. Do the partial "adds" as you go along. Here's an example; to multiply:

$$\begin{array}{r} 814 \\ \times \quad 84 \\ \hline \end{array}$$

follow this outline:

Step 1: Multiply 814 by 4 digit by digit; the result is 3256.

Step 2: Move in by one digit and start multiplying again. 8*4 gives a 2 with a carry of 3. Add the 2 to the 5 and the 3 to the 2. Total is now 3576. Now multiply the 8 by the 1 and add it to the 5, getting 3 with a carry of 1. Add this to get 4376. Finally, multiply the 8*8 to get 64. So add 4 to the 4 getting 8 and carry the 6. The final answer is 68376.

The tricky thing about programming this is keeping track of the various stages that tell you where you are in the process. I'll leave it to you. There are other methods that computer scientists have invented that are more complicated but this one is actually one of the better ones. I should also point out that besides saving time this method also saves space.

5
CHAPTER
Procedures

This chapter shows you how to use QuickBASIC's *SUB procedures*, or subprograms, as they're often called. A SUB procedure is a smaller helper program that you call as needed. Procedures generalize and extend the FUNCTION procedures of the last chapter. You'll also see how error trapping works.

Procedures are the last major addition to a QuickBASIC programmer's toolbox. When you master this material you will, in a very real sense, have finished learning how to program in QuickBASIC. From that point on it's just a matter of learning new commands and new ways of organizing your data.

Finally, although to this point I've mostly avoided generalities on how to write programs, I don't think it's a good idea to continue doing so. As programs grow longer, the possibilities for mistakes, for bugs, probably grows even faster. Since you are close to mastering all the tools of a master QB programmer, it's a good idea to see the methods that programmers have developed to make writing longer programs easier, or at least less frustrating. This is why I'll begin the chapter with a section on structured programming.

Modular—top down programming

The method described here, modular, top down, structured program design, developed from rules of thumb that programmers learned through experience. Don't take them as gospel. Ultimately, you will develop your own style for writing programs, and what works for one person may not work for another. Still, just as artists benefit from knowing what techniques have worked in the past, programmers can learn from what programmers have done before them.

The first rule is: "Think first—code later," sometimes expressed as, "The sooner you start writing, the longer it takes." You have some idea of what needs to be done, and so you enter the View window and start writing code. When your first attempt doesn't work you keep on modifying your program until it does, or seems to, work. This is usually referred, sometimes with pride, sometimes with disdain, as "hacking away at the keyboard."

Of course almost everyone will occasionally write programs with little or no preparation; that's one of the virtues of BASIC. Where do you draw the line? How long must a program be before it can benefit from some paper and pencil? Know your own limits. I find it hard to write a program longer than one screen, 20 to 25 lines, without some sort of outline. Whenever I try, even when I eventually get the program running, it ends up taking longer than if I had written an outline first.

Outlines don't have to be complicated. My complete outline for the pig latin program was:

```
While there are still words
    pig the next word
To pig a word
    one letter words stay the same
    beginning vowels -> add way
    beginning consonants (y, qu = conson) -> ROTATE and add ay
ROTATE
    find conson
    move it to end
```

Perhaps this may be a little hard for anyone but me to use, but that's not the point of an outline. My outlines are for me, yours are for you. In particular, my outlines help me fix the concepts that I'll use in the program, yours should help you. I find that a good rule is that when it looks to me like a line in my outline will correspond to no more than 10 lines of code then I've done enough outlining and I should start writing. Of course, only practice will let you see at a glance how long the coded version is likely to be.

Some people like to expand their outline to pseudo-code. This is especially common if you are developing a program with, or for, someone else. Pseudo-code is an ill-defined cross between a programming language and English. While everyone seems to have their own idea of what pseudo-code should look like, most programmers do agree that a pseudo-code description, unlike an outline, should be so clear and detailed that any competent programmer can translate it into a running program. Here's a pseudo-code version of part of the above outline:

```
Function(pig NEXT WORD)
```

```
        IF Length(NEXT WORD) = 1 THEN do nothing
        IF FirstLetter (NEXT WORD) = a,i,e,o,u THEN
            PIG(NEXT WORD) = NEXT WORD + WAY
    ELSE
        Find(leading consonants of NEXT WORD)    ' (qu a consonant)
    PIG(NEXT WORD) = NEXT WORD - leading consonants + ay
```

The point is that though the phrase:

```
    PIG(NEXT WORD) = NEXT WORD - leading consonants + ay
```

doesn't seem on the surface all that close to QuickBASIC, it is for an experienced programmer.

Think about writing a program this way: when you have something hard to do, you first divide it into several smaller jobs. Moreover, with most jobs, the "sub-tasks"—the smaller jobs—have a natural order in which they should be done. You dig a hole for the foundation before you call the cement truck.

Write programs from the general to the particular. Start with a conception of the big picture, and then, in stages, break it down. This lets you keep track of the forest, even when there are lots of trees. Your first outline lists the jobs that have to be done. Keep on refining your outline until the pieces to be coded are well within your limits. Stop massaging the problem when you can shut your eyes and see the code that does it. Sometimes this "step-wise refinement" is described as relentless massage. Since programmers often say "massage a problem" when they mean "chew it over and analyze it," the metaphor is striking—and useful.

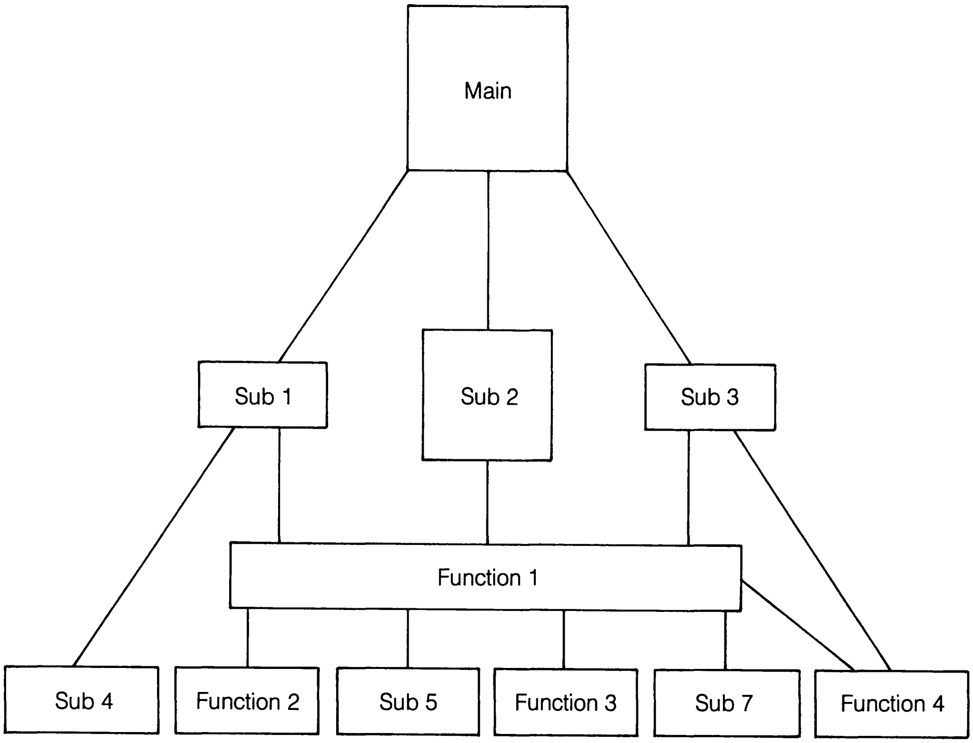
In a way, top-down design is like what high school teachers push at you as the right method to write papers—extensively outlining that term paper, before you put pen to paper. And, the reasons are the same: there's a limit on the number of balls you can juggle—there's a limit on the number of ideas you can deal with at once.

Some people find a chart like the one in Fig. 5-1 helpful. The placement of the boxes indicates how the jobs relate. Each box usually will correspond to a SUB procedure or user-defined FUNCTION procedure. SUB Procedures and FUNCTION procedures on a higher level use only the results of procedures and functions on the same or lower level.

Programmers often call these building blocks modules—that's why what we're doing is called modular programming. Unfortunately, there's another more technical meaning of module in QuickBASIC—you'll see more about that in chapter 9.

Even if you can see how to program two completely separate jobs in one SUB or FUNCTION, it's usually better not to. Sticking to one job per procedure makes it easier to both debug the procedure and to optimize the code in it.

Most of a professional programmer's time is spent modifying programs written by other people. Imagine a big program that wasn't written top-down: no distinction was made between local and global variables—all variables are global; no



5-1 A programming diagram.

attempt to write the program in digestible pieces with clear lines of communications between the pieces.

What happens? Because all variables are global a little change you make could foul up the whole program. Because the pieces aren't digestible and the way they communicate is unclear you can't be sure how they relate. Anything you do to one line may introduce side effects—possibly disastrous ones.

This kind of disaster was common until the late 1960s or early 1970s. Companies first spent millions of dollars having programs modified; then they spent more money trying to anticipate the potential side effects the changes they just paid for might cause. And then they hoped that more time (and more money) would fix the side effects. No matter where they stopped, they never could be certain that the programs were free of bugs. Top-down design, when combined with programming languages that allowed local and global variables, can stop side effects completely: programs still have bugs, but these bugs don't cause epidemics. If you fix a small module in a giant top-down designed program, then you know how the changes affect everything.

Finally, it's worth pointing out that QuickBASIC is a modern BASIC, so it was designed from the start to make modularizing a program possible, and, in fact, not all that difficult. Interpreted BASICs lack local and global variables as well as the multi-line function and sub-programs that you need. For this reason alone, even if it were not many times faster, I would have switched to QuickBASIC.

Getting started with procedures

You have already seen one method to modularize a program: a FUNCTION procedure. As you saw in the last chapter, a QuickBASIC FUNCTION can be made to do most anything, provided that what you want to do is get an answer out of it. As I mentioned there, although functions can make any kind of display that you want, it's not a good idea to do so unless the display is somehow related to what the function is designed to do. In any case, a function takes raw data, massages it, and then returns a single value. In particular, although you can write a function that will work through a list as it's entered and find the smallest or the largest value, it can't find both: a function can never sort a list. You'll see many ways to do this in the next two chapters.

Suppose, for example, you wanted to print a song, one with many verses but only a single chorus. The outline is clear:

```
While there are verses left
    print verse
    print chorus
loop
```

Unless you wrote a truly bizarre function, you could not easily translate this outline into QuickBASIC using a FUNCTION. To do this you need a new structure: the procedure, or sub-program.

The structure of the simplest kind of procedure—although one powerful enough to translate the outline, looks like:

```
SUB chorus
    many print statements
END SUB
```

The first line has the key word SUB, followed by the procedure's name. A procedure name can be up to 40 characters. Next comes the statements that make up the procedure. Finally, there is the key word END SUB to indicate the end of the procedure.

Just as for FUNCTIONs, QuickBASIC keeps all your SUB-programs in separate windows. And, just as for functions, you can cycle through them via SHIFT + F2.

Thinking of a program as a song with many choruses would be a good metaphor for a program and its procedures and functions, except that it misses one key point about using a procedure: each time you need the procedure it's likely to be in a different situation. The procedure must change to meet new requirements. We need a way to transfer information between the main program and the procedure. You do this in much the same way as you did for functions: you give the procedure a parameter list. This parameter list is used to communicate between the main program and the procedure. For example, suppose you wanted to print the old song, "100 bottles of beer on the wall."

> 100 bottles of beer on the wall,
> 100 bottles of beer,
> If one of those bottles should happen to fall,
> 99 bottles of beer on the wall.
>
> 99 bottles of beer on the wall,
> 99 bottles of beer,
> If one of those bottles should happen to fall,
> 98 bottles of beer on the wall.
>
> 98 bottles of beer on the wall,
> 98 bottles of beer,
> If one of those bottles should happen to fall,
> 97 bottles of beer on the wall.

etc.
Here is a program to print the verses of this song using a procedure.

```
DECLARE SUB Chorus (X%)
'CH5\P1.BAS
' A drinking song

DEFINT A-Z

FOR I = 100 TO 1 STEP -1
  CALL Chorus(I)
NEXT I

END

SUB Chorus (X)

  PRINT
  PRINT X; "bottles of beer on the wall,"
  PRINT X; "bottles of beer,"
  PRINT "If one of those bottles should happen to fall,"
  PRINT X - 1; "bottles of beer on the wall."

END SUB
```

On each pass through the loop the variable I is sent to the procedure, where it replaces the formal parameter X. Here the I is also called the actual parameter, as was the case for functions. Also just like with functions the names you choose for your formal parameters are irrelevant; they just serve as placeholders. Note as well the DECLARE statement, which must precede all statements. SUB-programs, like FUNCTIONs, require DECLARE statements to keep track of the type of parameters. And, as before, QuickBASIC automatically generates the appropriate DECLARE statements when you save the program. Some people frequently SAVE a program while developing it just to see what the DECLARE statements look like. This can help you figure out what types of variables QuickBASIC thinks it will be handling. (A good way to make sure that you haven't put in or left out a DEFiner.) Finally, note that you do not use a type identifier for a sub-program.

You may be saying that this particular example seems a little forced: it's easy to re-write the program using a FOR-NEXT loop. Of course this is true, but writing the program using a procedure or sub-program changes the emphasis a little—it's a lot closer to my outline. And, imagine a FOR-NEXT loop that had to surround 50 lines of code. In this situation its too easy to forget what the loop is doing: most programmers prefer loops to be digestible—the whole loop better not involve more than a single screen of code, in my case.

For a more interesting program consider a frame builder. A long program might want to put boxes around text in many different contexts. Here is a procedure for this:

```
DECLARE SUB BoxDrawer (Rw%, Col%, Hsize%, Vsize%)
DEFINT A-Z

SUB BoxDrawer (Rw, Col, Hsize, Vsize)
' A rectangle drawer
' this procedure accepts a row value
' a column value and a horizontal and vertical size
' it doesn't check the data.                              'A

' LOCAL variables are: TopLftCor$, TopRtCor$,BotLfCor$,BotRtCor$
' Start,VertFinish,EndCol                                 'B

TopLftCor$ = CHR$(218)                                    'C
TopRtCor$ = CHR$(191)

BotLfCor$ = CHR$(192)
BotRtCor$ = CHR$(217)

HorizLine$ = STRING$(Hsize - 2, 196)
VertLine$ = CHR$(179)


Top$ = TopLftCor$ + HorizLine$ + TopRtCor$
Bot$ = BotLfCor$ + HorizLine$ + BotRtCor$

LOCATE Rw, Col
PRINT Top$;

Start = Rw + 1
```

```
VertFinish = Rw + Vsize - 2
EndCol = Col + Hsize - 1

FOR I = Start TO VertFinish

   LOCATE I, Col
   PRINT VertLine$;
   LOCATE I, EndCol
   PRINT VertLine$;

NEXT I

LOCATE VertFinish + 1, Col
PRINT Bot$;

END SUB
```

A) Just like with a FUNCTION, it's best to check the values that you send a procedure before you CALL the procedure—that's why I've eliminated the checks that were in the original program.
B) Again, just like the functions, we should initialize the local and shared variables before any executable statements.
C) Here you might prefer to allow double sided boxes. To do this, add another parameter, say Type, to the procedure, and re-write it as:

```
SUB BoxDrawer(Rw,Col,HSize,Vsize,Type)
    SELECT CASE Type
        CASE 1
            TopLftCor$ = CHR$(218)
            TopRtCor$ = CHR$(191)
            .
            .
            .
        CASE ELSE
    TopLftCor$ = CHR$(201)
    TopRtCor$ = CHR$(187)
            .
            .
```

etc.

Once the procedure is written and debugged then you can use it whenever you want, in whatever program needs it.

For example, to box some text that you're about to print on the screen, use the CSRLIN and POS(0) functions before and after you print the text. Here is a program that gives a demonstration of this:

```
'CH5\P2.BAS
' A demo using the Box procedure
```

```
CLS
DEFINT A-Z

PRINT "Press any key to see the box procedure at work"
I$ = INPUT$(1)
LOCATE 12,21
TestString$ = "A Procedure a day keeps the bugs away"
PRINT TestString$

Row = CSRLIN -2                     'A
Column = 20
HorizSize = LEN(TestString$) + 2            'B
VertSize = 3

CALL BoxDrawer(Row,Column,HorizSize,VertSize)

END
```

A) The CSRLIN function gives you the current line the cursor is on, which is two lines down from where we want the box to start.
B) The box has to be two character longer than the string.

When you start writing procedures in your programs is immaterial; my own preference is to do them after I've written all the module level code. (Which is a fancy term for what doesn't have its own window.) Presumably in my outline I've marked out what parameters they'll take, but QuickBASIC doesn't mind. On the other hand, you can't run a program with a CALL to an unwritten procedure. For more on this, see the last section.

As a final example for this section, I will do a bar graph program, using a group of procedure calls. This means that you'll use separate procedures to draw the axes and to display an individual bar. By doing this you can easily display as many bars as fit on a single screen, say 25, if you space them three apart.
This is the main module:

```
DECLARE SUB DrawAxes ()
DECLARE SUB DrawBar (Where!, HowHigh!)
DECLARE FUNCTION GetValue ()

'CH5\P3.BAS


CLS
PRINT "This program draw up to a 25 bar, bar graph.  You"
PRINT "can enter up to 25 numbers between 1 and 23."
PRINT
PRINT "Press any key to start."
I$ = INPUT$(1)
CLS

CALL DrawAxes
```

```
BARS = 1

Height = GetValue                               'A

DO UNTIL Height = 0

   Location = 3 * BARS
   CALL DrawBar(Location, Height)               'B

   BARS = BARS + 1
   IF BARS > 25 THEN EXIT DO                    'C

   Height = GetValue

 LOOP

LOCATE 25, 1
PRINT SPACE$(79);
LOCATE 25, 1
PRINT "Press any key to clear screen";
I$ = INPUT$(1)
CLS

END
```

A) This function, which accepts the values for the heights of the bars, will need to be done carefully (see below). After all, we don't want to disturb what is already displayed in the graph. And we have to make sure the values are permissible.
B) The procedure to draw a bar will obviously use two parameters: where to draw the bar, and how high to make it. Because you want to space the bars three spaces apart, you can use three times the number of the bar for the where. How high comes from the as yet unwritten function GetValue.
C) As I mentioned earlier, use an EXIT LOOP only for abnormal methods of ending a loop. This qualifies because the natural way to end the program will be to enter a value of zero for the height of a bar.

That takes care of the main program. Notice how most of the gritty details have been pushed under the rug. This is quite common when you use procedures: your main programs will have a fairly clean look, containing directions and repeated procedure and function calls. In fact, some people would even put the directions into procedure calls and so make the main module into one long sequence of procedure and function calls. This too is a matter of taste. In any case, it's unlikely that the main module will need to be very long.

Here is the procedure to draw the axes:

```
SUB DrawAxes

' LOCAL variables are: TickMarks$,BoxChar$,I
```

```
TickMark$ = CHR$(95)
BoxChar$ = CHR$(219)


FOR I = 1 TO 22
  LOCATE I, 1
  PRINT BoxChar$ + TickMark$;
NEXT I

LOCATE 23, 1: PRINT BoxChar$;
LOCATE 24, 1: PRINT STRING$(78, BoxChar$);

END SUB
```

Next is the procedure to draw a bar. As mentioned before, unlike the previous procedure which didn't use any parameters (which is why the DECLARE statement used an empty set of parentheses), this one depends on two parameters, "Where" and "HowHigh."

```
SUB DrawBar (Where, HowHigh)

' LOCAL Variables are: TopOfBar,BoxChar$,I

BoxChar$ = CHR$(178)
TopOfBar = 24 - HowHigh

FOR I = TopOfBar TO 23
  LOCATE I, Where
  PRINT BoxChar$ + BoxChar$;
NEXT I

END SUB
```

Finally, we need to write the function that gets the data. An input routine is often the most subtle part of a program, and here is no exception. Bulletproofing input often requires more lines of code then on first glance seem necessary. For example, in what follows I've decided to mask out all but numbers. I do this by accepting the number digit by digit, disregarding any other characters. As mentioned before, you can do this by using the INPUT$(1) command. Once all the digits are in, you use the VAL command to convert it to a number. You saw a version of this kind of routine earlier.

This function is further complicated by the need to check that the values are not too big and the requirement that you not disturb the bars already on the screen. The easiest way to do the former is to use the 25th line of the screen for the data. This means you have to prevent the screen from scrolling. For the latter, you'll write the main part of the function as a loop that only ends when a number between 0 and 23 is entered.

This is the function:

```
FUNCTION GetValue

' Draw axes
' modified from CH5\P19.BAS

' LOCAL variables are: Temp$, TempValue,Dig$


DO                                                          'A

LOCATE 25, 1                                                'B
PRINT SPACE$(79);
LOCATE 25, 1
PRINT "Type a number between 1 and 23 (zero to end) and hit ";
PRINT "enter when done.";
LOCATE 25, 1

Temp$ = ""
Dig$ = INPUT$(1)
PRINT SPACE$(79);                                      'C

LOCATE 25, 1

  DO UNTIL Dig$ = CHR$(13)

    IF Dig$ < "0" OR Dig$ > "9" THEN
      BEEP
    ELSE
      PRINT Dig$;                                      'D
      Temp$ = Temp$ + Dig$
    END IF

    Dig$ = INPUT$(1)

  LOOP                                         'stops after enter key

  TempValue = VAL(Temp$)

LOOP UNTIL TempValue >= 0 AND TempValue <= 23            'E

GetValue = TempValue

END FUNCTION
```

A) This loop will make sure that the number is acceptable.
B) I'll use the 25th line of the screen to accept data and give some simple directions. However, I have to be especially careful not to allow the screen to scroll. VIEW PRINT 25 TO 25 is another way to do this.
C) This erases the previous prompt.
D) I only print the digit if I'm sure that it is acceptable. In particular, the program never prints a carriage return, and so the display never scrolls.

E) If you enter too large a number, then the loop begins again and the string variable Temp$ is re-initialized.

You may be wondering why did I use a function here; why not a procedure, SUB GetNumber, with a shared variable "HowHigh"? This is just a matter of taste. As you have probably realized, a procedure can do anything a function can do, but I feel it's a good idea to preserve the distinction. I use functions when I need to massage values and/or make new ones: I use procedures in all other situations. For me, FUNCTIONs massage and sub-programs do.

Finally, let me end this section by noting that, just as for FUNCTIONs, you can use STATIC variables in sub-programs. And, also just like for FUNCTIONs, they make excellent debugging tools. Since procedures, like FUNCTIONs, that use only STATIC variables run faster, QuickBASIC lets you force all variables to be STATIC by writing:

```
SUB . . . . . . . . . . . . . . . STATIC
```

More advanced use of procedures and functions

Consider the following simple FUNCTION:

```
FUNCTION INCR(X)
    X = X + 1
END FUNCTION
```

or

```
SUB ADD1(X)
    X = X + 1
END SUB
```

This is a new situation—I've assigned a new value to the formal parameter. However, not only is there no reason not to make assignments to the parameters in procedures and functions, but you'll never fully exploit the power of procedures until you become comfortable with what happens when you do.

When you assign values to a formal parameter in a procedure or function then, after you call the procedure or use the function, the value of the corresponding variable changes accordingly.

In particular, with the procedure given above consider:

```
Number = 5
CALL ADD1(Number)
PRINT Number
PRINT ADD1(Number)
```

This will display a 6 and a 7 on the screen.

Think about it this way: when you call a procedure or a function, then QuickBASIC replaces the placeholder with the variable, not with the value of the variable, as it does with the older DEF FN functions. In the above example, QuickBASIC sends the variable Number to the procedure, where it replaces the placeholder X—everywhere X occurred. (Actually it sends the address in memory where the value of Number is stored.) Similarly, the second time QuickBASIC sends the address of the variable its current value is a 6, so it adds 1 and comes up with 7, which is now the current value of Number. In particular, calling the procedure with the statement CALL ADD1(Number), or using the function, is the same as saying:

```
Number = Number + 1
```

because the variable Number substitutes for the placeholder X everywhere the variable X appeared.

As another example, suppose you needed to change a phrase by stripping out any spaces in a phrase, i.e., a super version of the function LTRIM$. The outline is clear:

```
Find out where the spaces (=CHR$(32) ) are and
remove them.
```

You can use the MID$ command to extract the non-space character in the string. The code that strips out spaces is easy:

```
For Count = 1 TO LenPhrase
    IF MID$(Phrase$,Count,1) < > Chr$(32) THEN
        Temp$ = Temp$ + MID$(Phrase$,Count,1)
NEXT Count
```

When this loop ends, the value of Temp$ is the stripped phrase. Now you have to decide: do you want to change the phrase or set up a new phrase? If you want the original phrase to change, incorporate the fragment given above into a procedure:

```
SUB StripSpaces(X$)

' LOCAL variables are Count,LenPhrase$,Temp$

LenPhrase = LEN(X$)

FOR Count = 1 TO LenPhrase
 IF MID$(X$,Count,1) <> Chr$(32) THEN
   Temp$ = Temp$ + MID$(X$,Count,1)
NEXT Count
X$ = TEMP$
END SUB
```

Now whenever you call the procedure:

    CALL StripSpaces(Phrase$)

the original string Phrase$ changes. If you made a function out of it:

```
FUNCTION StripSpaces$(X$)

' LOCAL variables are Count,LenPhrase$,Temp$

LenPhrase = LEN(X$)

FOR Count = 1 TO LenPhrase
 IF MID$(X$,Count,1) <> Chr$(32) THEN
   Temp$ = Temp$ + MID$(X$,Count,1)
NEXT Count
StripSpaces$ = TEMP$
END FUNCTION
```

Then the value of this function, StripSpace$(Phrase$), is the stripped phrase but the original Phrase$ is still intact. Or you can add a line to the function to make it equivalent to the sub-program:

    X$ = Temp$

The reason using a procedure or function with a variable is called passing by reference is because the compiler sends the procedure the location in memory that the variables take up. Since the procedure or function now knows where the values of the variables are, it can change them. Of course, this can only happen if you make an assignment to a formal parameter within it.

When you call an older DEF FN function, QuickBASIC copies the values of the variables but doesn't send the location, so this type of function can't change anything.

Because procedures and functions can change the values of the variables used as actual parameters, any time you call a procedure, you are, in a sense, defining a new group of SHARED variables—the variables you just sent as actual parameters.

How do you decide whether to make a variable SHARED, or send it as a parameter? Most programmers follow the convention that shared variables are for global information (like the value of π), and therefore you should not change the value of shared variables in a procedure. Procedures should only change the values of parameters. The reason for this convention stems from the methods you'll use to debug procedures, for which see the next section.

Occasionally, you'll want to send procedures values of variables, or even numeric expressions, numbers or calculations, in which case they won't change. This is called passing by value, and is exactly what happens with DEF FN functions. To pass information by value to a procedure, enclose the parameter in parentheses. For example, consider the following test program:

```
SUB ADDI(X)
 X = X + 1
END SUB

Number = 5

CALL ADD1((Number))                    'NOTE THE EXTRA PARENTHESIS!
PRINT Number

'compare this to

CALL ADD1(Number)
PRINT Number

END
```

What you'll see is:

```
    5
    6
```

You see a five and then a six because, in the first call to the ADD1 procedure, we are passing the variable Number by value—the procedure can't change the variable. In the second case, we're passing by reference—the procedure can and does change the value of Number—it makes it a 6. Essentially, the first call is a complicated way of saying:

```
    CALL ADD1(5)
```

You should also be aware that because procedures, in a sense, deal with variables—when you pass by reference—or with values or constants—when you pass by value—the QuickBASIC manual recommends that you do not pass the same variable by reference for more than one parameter. Don't say CALL BadExample(X,X). This leads to what is called variable aliasing, and it can lead to extremely subtle bugs. Aliasing means two symbols are referring to the same place in memory.

As a final example for this section, suppose you wanted to write a program to create and then shuffle a deck of cards. You can think of a deck of cards as being a string variable:

```
    DeckOfCard$ = "2♣3♣4♣5♣6♣7♣8♣9♣0♣J♣Q♣K♣A♣ . . ."
```

where I've used 0 for the 10 to keep all the cards the same length. Obviously, because this string variable will have 104 characters, you'll want the computer to do some of the work. You'll see one slightly awkward method now, and a much better one in the next chapter.

In any case since we want to change this variable when we shuffle the cards we should write a:

```
SUB Shuffle(X$)
```

procedure, and not a function. This procedure will be similar to the Jumble program (CH3 \ P13.BAS). The only difference is that you have to move characters by two's; that's why I used a 0 for the 10. Here is the jumble program rewritten as a procedure to do this:

```
DECLARE SUB Shuffle (Phrase$)
DECLARE FUNCTION Suit$ (X%)

' a Jumble program (CH2\P13.BAS)
' converted to a procedure
' jumbles by twos

SUB Shuffle (Phrase$)

' LOCAL variables are: LenPhrase,I

RANDOMIZE TIMER
LenPhrase = LEN(Phrase$) / 2                          'A

FOR I = 1 TO LenPhrase

  INum = INT(LenPhrase * RND(1)) + 1
  NewChar$ = MID$(Phrase$, 2 * INum - 1, 2)           'B
  OldChar$ = MID$(Phrase$, 2 * I - 1, 2)

  MID$(Phrase$, 2 * I - 1, 2) = NewChar$               'C
  MID$(Phrase$, 2 * INum - 1, 1) = OldChar$

NEXT I

END SUB
```

A) The number of cards is half the length of the phrase.
B) The cards are located starting in the 1,3,5,7 . . . positions.
C) You need to move two characters at a time.

Next you need to set up the deck of cards. By consulting, for example, the ASCII codes in the on-line help, you'll see that:

Clubs symbol	(♣)	=	CHR$(5)
Diamond symbol	(♦)	=	CHR$(4)
Hearts symbol	(♥)	=	CHR$(3)
Spades symbol	(♠)	=	CHR$(6)

The easiest way to use this information is with a function. This is a function that makes a suit of cards.

```
FUNCTION Suit$ (X%)

 'LOCAL variables I,Temp$

 Temp$ = ""

FOR I = 2 TO 10
  Temp$ = Temp$ + RIGHT$( STR$(I), 1) + CHR$(X%)
NEXT I

HighCards$ = "J"+CHR$(X%)+"Q"+CHR$(X%)+"K"+CHR$(X%)+"A"+CHR$(X%)
Suit$ = Temp$ + HighCards$

END FUNCTION
```

Notice that if I = 10, then RIGHT$(STR$(I),1) = "0." Now you can call this function four times:

DeckOfCards$ = Suit$(5) + Suit$(4) + Suit$(3) + Suit$(6)

Now that you've done all the preliminaries, next is a procedure that deals out and displays a bridge hand:

```
SUB Deal

' LOCAL variable is Cards

SHARED DeckOfCards$

CALL Shuffle(DeckOfCards$)

  For Cards = 1 TO 52
    PRINT MID$(DeckOfCards$,2*Cards-1,2)              'A
  NEXT Cards

END SUB
```

A) Again the cards start in the first, third, fifth etc. positions.

Passing variables by references versus passing them by value is a subtle concept. However, the distinction is important and it's necessary to understand it for anyone doing structured programming. Knowing when to pass variables by value as opposed to passing them by reference is one of the marks of a master programmer. In the chapter on recursion you'll see a couple of more realistic examples of why this distinction is so important.

Of course, if you never make assignments to the parameters during the course of a procedure or function then there's no real difference other than memory use. Passing by value, because it involves copying the value of the variable, uses slightly more memory.

Testing and debugging programs

No matter how carefully you outline your program, after you write it you will still need to test it. Some people's idea of testing consists of RUNning the program a few times, each time using slightly different inputs to see what happens. This can work out well when you have a short program but it's not effective or convincing for a long program, or even a short program that is in any way subtle.

For a sophisticated or long program you should not only write the program top down, you should also, as much as possible, test it from the top down. This means that as you finish each procedure or function, combine it with the pieces you've already checked, and test everything again. Of course, a higher level procedure or function may need results from a piece not yet written. In this case the best technique, often called stub programming, is to substitute constants where necessary for the results of as yet unwritten procedures or functions. Define the SUB procedure or FUNCTION procedure but fill it with constants instead of having it do anything. The procedure calls will still work the same, but they get only these constants, the test results, back from the "stubs."

Testing programs is an art, not a science. There's no rule that always works. If your program is long and complicated, you can't test all the possibilities—you have to be content with reasonable ones.

The key is the word reasonable, and the following apocryphal story explains how subtle this concept can be. A utility company had a complicated, but, they thought, carefully checked program to send out bills, follow up bills, and finally to automatically cut off service if no response was received. Well, one day, the story goes, someone went on vacation and shut off the electricity. The computer sent out a bill for $0.00, which, understandably, wasn't paid. After the requisite number of follow-up requests, the computer finally issued a termination notice saying that if this unfortunate person didn't pay $0.00 by Thursday his electricity would be automatically cut off.

A frantic call, if he managed to get through to a real live person, may have succeeded in stopping the shut-off; the story doesn't say. If this story is true, then the programmer forgot to test what the program would do if the bill was $0.00. This wasn't a reasonable possibility. Of course all they had to do is change a >= to a >, somewhere in the program.

Finally, remember that regardless of how much you test, you can never be completely sure that your program won't crash. From experience I know that no matter how robust I try to make a program, it always seems that someone, somehow will find a way to crash it. However, my goal has always been to write programs which conform to a sign that I once saw. Slightly paraphrased it said:

Our goal is a program:
THAT SPUTTERS OUT AND DOESN'T BLOW UP!

The technical term for this is to degrade gracefully. A realistic goal is not a perfect program but one that is as robust as possible. Check the data you send to functions and procedures before you call them. Check what the user enters before the program uses it. Better yet, vet the data while it's being entered.

Anyway, before you can bulletproof a program you need one that at least seems to work. So you've compiled the program and know there are no syntax errors—but you've tested the program and it doesn't work. There are bugs to isolate and eradicate. Don't be surprised or dismayed—bugs come with the territory. You have to find them and determine what kind they are. There are actually two kinds of bugs: logical and grammatical. The grammatical errors are things like a misspelled variable name, leading to a default value that ruins the program. Surprisingly enough they are often the most difficult kind of bug to detect. The best cure is a programmers tool called a cross referencer or XREF program. This program works through the source code of a program and then lists the names of all the variables that occur, along with where they occur. You'll see how to write a version of an XREF program in chapter 7.

QuickBASIC actually has a primitive form of XREF built in. If you place the cursor at a variable and hit F1, the "help on" key, then QB gives you a list of all procedures and functions that use that variable. An XREF program on the other hand gives you a listing of all the variables and complete details on every occurrence of every variable. F1 can only work one variable at a time, making it much harder to find a misspelled variable name.

In any case you can always isolate this kind of typo by using the techniques described below, and the watch facility you've already seen.

To get rid of a more subtle, logical bug, you have to isolate it: find the part of the program that's causing the problem. If you've followed the modular approach, then your life is a whole lot easier. If you've been top down testing the program as you develop it, then it's clear where the problem is.

So, pinpointing the procedure or function that's causing the problem isn't usually hard, if it's your program. Mostly because you start off with a good idea of the logic of your program. If it's not your program, or you waited until the program was finished, then you can use the techniques below to check the pieces one at a time. If the program wasn't modular then you're on your own. I don't know any way to check an unstructured program, other than to work through it line by line. And it will often be faster to rewrite it in a more structured fashion.

Let's assume that you've decided on the procedure or function that's causing the trouble. There are only three possibilities:

1. What's going in is wrong; what you've fed to the procedure or function is confusing it.
2. What's going out is wrong; it's sending incorrect information to other parts of the program.

3. There's something wrong inside the procedure or function. For example, it doesn't clear the screen at the right time.

In the first two cases, the fault is either in the parameters you send to the procedure or function, what you've assigned to the parameters, or the SHARED variables within the function or procedure.

How do you decide which? Well, it's hard to imagine a correctly written procedure or function where you couldn't analyze on a piece of paper what should happen in many cases. Work through the procedure or function by hand, playing "computer." This means, don't make any assumptions other than what the computer would know at that point. Don't assume variables have certain values unless you can convince yourself that they do.

Write a *driver* program, using the Immediate window. A driver is a program fragment that calls a function or procedure with specific values. For example, suppose you know that when global variables var1 = 10 and var2 = 20, the results of a procedure (a global variable—say global1) has value 97. When you want to test how this procedure or function behaves at a particular place in a program, add a breakpoint, and use the Immediate window to enter:

CALL *WhateverYou'reTesting*(10,20)
PRINT "The value of the variable global1 is ";global1

See what happens. If the value of the variable global1 isn't right, then use the Debug menu to watch the important variables, the ones that affect global1. However, you can only watch variables within a procedure or function if you've added them to the watch list from within that function or procedure. There may be something wrong inside the procedure or function. If the value of global1 is right, then determine (again by hand) what happens for some other values. Try the boundary values, the strange values, like the $0.00 that the programmer in the story forgot. If you always match up with what you expect, then there's probably nothing wrong with the procedure or function. You now expect that the problem comes from outside. Of course, in practice you have to make sure your driver fragment sends all the information needed by the procedure or function. And, that's not likely to be only two global variables. You may not want to use the Immediate window then—you might prefer to insert the code at the breakpoint.

Check each procedure that calls this procedure or function—apply the same techniques for them—check what goes in and out of these procedures or functions. QuickBASIC makes this easy: not only can you single step through a program, you can step through a procedure or function. When you're debugging a program, or using the stub programming method mentioned earlier, you can use F10 rather than F8 to single step. The important difference is that F10 treats a call to a FUNCTION or sub-program as one step. This way you don't have to step through all the lines in all the functions and procedures in your program, when you don't need to.

In every case, then, you eventually wind your way down to a procedure or function that just doesn't work.

You now know that you have an error internal to a procedure or function. To detect these errors, try these techniques:

1. Add more (and more) variables to watch.
2. Single step through the procedure using Restart, breakpoints, and the F8 or F10 key as needed. While you're doing this, toggle back and forth to the output screen to see if what's there is what you expect.

Feeding a procedure or function specific numbers, and using these techniques, is not a cure-all. No technique can help unless you have a good grip on what the procedure or function should do. If you use an IF-THEN, are you testing for the right quantity? Should a ">=" be a ">"? Watch the value of any Boolean relations that seem to be off. Check any loops that are inside the routine—loops are a common source of problems. Are counters initialized right (do you have an "off-by-one" error)? Are you testing your indeterminate loops at the top, when you should do it at the bottom? All these can be checked by Watching the appropriate object, like an inequality: X>=0.

Documentation, managing procedures and program style

Although you can remember the logic of a complicated program for a while, you can't remember it forever. Good documentation is the key that can open the lock. Some people include the pseudo-code or outline for the program as multiple remark statements. Along with meaningful variable names, this is obviously the best form of documentation. Try to avoid tricky code—if you need to do something extraordinarily clever, make sure it's extensively commented. And, most of the time, you'll find that the clever piece of code wasn't really needed. Nothing is harder to change six months down the line than cute code. Cute code often comes from a misplaced attempt to get a program to run faster. While sometimes necessary, QuickBASIC is usually fast enough so that this is more often an ego thing. I'm reminded of a sign I once saw:

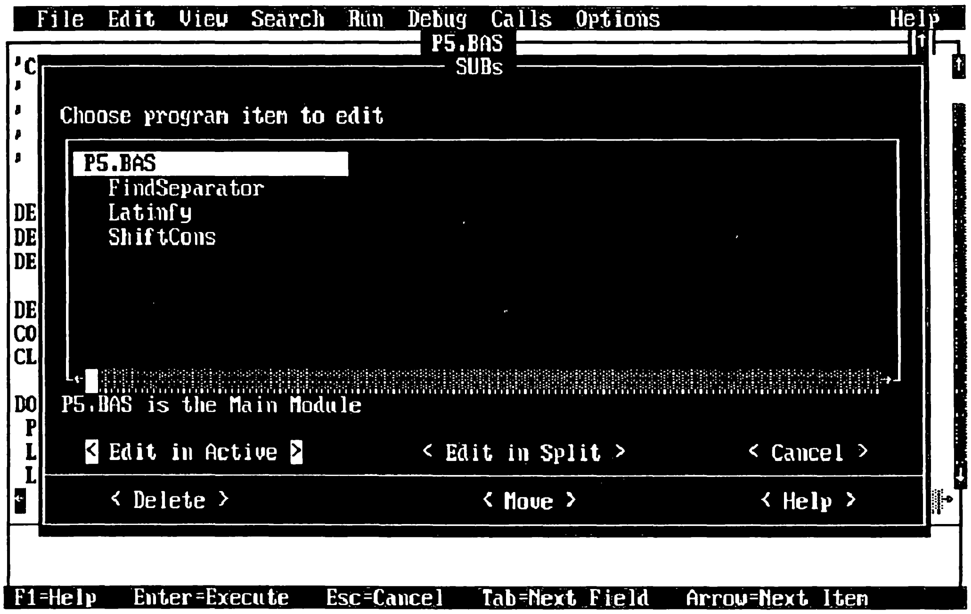
Rules for program optimization:
1. DON'T DO IT.
2. For expert programmers only, <u>DON'T DO IT!</u>

The point being that when you start thinking of tricks to speed up your programs then you can easily lose sight of the fundamental issue: making sure they run robustly in the first place.

In fact, dramatic speedups usually come from shifts in the *algorithms* in the program, not from quite little tweaks. An algorithm is the method you use to solve a problem. For example, the second method of programming an infinite multiplier from the last chapter is a better algorithm than the first, so a program using it would run much faster. Similarly, in problems like sorting a list, it's the method that you choose which determines how fast the sort is. In the next two chapters you'll see how choosing the right sorting technique can speed up a program 1000 fold. This is more than any minor tweak can ever hope to accomplish. Discovering new, hopefully faster algorithms is one of the main tasks of computer scientists and mathematicians.

In any case, it's extremely difficult to modify or debug a program, even one that you long ago wrote yourself, that has few or no remark statements, little accompanying documentation, and has cute or some other kind of non-informative variable names. A procedure called MakeMartini(Shaken, ButNot, Stirred) should be in a program about James Bond, not in a program about trig functions. Or, since QuickBASIC allows long variable names, don't make your programs a morass of variables named X, X13, X17, X39, etc.

Finally, as your programs become more sophisticated, they'll start using more and more procedures and functions—a program might have 15 sub-programs and 12 FUNCTIONs. Obviously, this is too many to cycle through using SHIFT + F2. Instead, you use a feature on the View menu. Open the View menu and hit S for Subs (shortcut is F2). This gives you a list of all the functions and procedures in your program. For example, Fig. 5-2 shows the one for the Pig latin program.


File Edit View Search Run Debug Calls Options Help
P5.BAS
SUBs
< Edit in Active >
< Edit in Split >
< Cancel >
< Delete >
< Move >
< Help >
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

```
Choose program item to edit

P5.BAS
   FindSeparator
   Latinfy
   ShiftCons

P5.BAS is the Main Module
```

5-2 The subs dialog box.

Notice that the names of the FUNCTIONs are indented slightly from the name of the main module. You can use this dialog box to start editing any one of the procedures, or the main module. All you have to do is move the highlighted bar to the piece you want to edit, and hit Enter. That's what Edit in Active means.

Finally, let me remind you again that chapter 11 of this book explains how you can build or buy a library of useful procedures and functions, and how to incorporate pieces of this library into your programs. If a procedure or function works well, there's a way to save it so that it's easy to incorporate into other programs. This is done from the full File menu.

In fact, a complicated program will often have many modules. Each module will have its own procedures and functions, possibly with a bit of module level code. These modules came up before in a slightly different context. This means that what you often end up doing in a programming project, is tieing together pieces of a thoroughly debugged library of sub-programs and functions by writing one main module. The big advantage to this is that you don't have to reinvent the wheel.

Error trapping

The command that activates "error trapping" is:

```
ON ERROR GOTO . . .
```

where the three dots stand for the label or line number that define the error trap. This command which can occur anywhere stops QB from bombing a program when an error occurs—obviously the ON ERROR command should transfer control to a piece of code that:

1. Identifies the problem.
2. If possible fixes it.

The "island of code" that defines the error trap must be in the module level code.

If the error can be corrected then the statement RESUME takes you back to the statement that caused the error in the first place. However, you can't correct an error if you don't know why it happened. You identify the problem by means of the ERR function. This gives you an integer that you can assign to a variable. For example, by saying:

```
ErrorNumber = ERR
```

then the value of the variable ErrorNumber can help you pick up the type of error. QB can identify more than 40 run-time errors. Appendix G of this book gives you the most useful one; appendix I of the Programming in BASIC manual has a complete list. For example:

```
25  Device fault. (For example, trying to LPRINT when the printer is off.)
27  Out of paper.
```

The way you use this information is simple. Somewhere in the program, before the error can occur, place, for example, the statement:

```
ON ERROR GOTO PrinterCheck
```

Now write a module labeled PrinterCheck, or whatever name you chose:

```
PrinterCheck:
    ErrorNumber = ERR
    BEEP
    SELECT CASE ErrorNumber
        CASE 25
            PRINT "Your printer may not be on"
        CASE 27
            PRINT "You're probably out of paper"
        CASE ELSE
            PRINT "Please tell the operator that"
            PRINT " error number ";ErrorNumber;"occurred."
    END SELECT
    PRINT "If the error has been corrected press 'Y' otherwise "
    PRINT "press any key to END"
    Continue$ = INPUT$(1)
    IF Continue$ = "Y" or Continue$ = "y" THEN RESUME ELSE END
```

The idea of this error trap is simple—the SELECT CASE statement is ideal. Each case tries to give some indication of where the problem is. And, if possible, how to correct it. If we reach the CASE ELSE then the error number has to be reported. In any case, the final block gives one the option of continuing or not.

Error trapping isn't a cure all. Obviously very little can be done about a hard disk crash, or the user not having any paper around. Moreover the PRINT statements in the error trap are likely to mess up the display, unless you use a different video page to write the message.

The RESUME command is actually quite flexible, because of two other variants. You can use:

```
RESUME NEXT
```

which sends processing to the statement following the one that caused the error. Or you can use:

```
RESUME label
```

which sends processing to the statement identified by the label or line number.

Occasionally, when debugging a program, it's helpful to know where the error occurred. This is done using the Break on Errors option from the Debug menu. This halts execution when an error occurs. After this you can use History back (= SHIFT+F8) to move back from the breakpoint.

ERRDEV$ is occasionally useful. This gives you the name of the device that caused the error. Its values can be:

A:	B:	C:	D:	E: (the disk drives)
LPT1:	LPT2:	LPT3:	PRN:	(the three printer ports and the generic printer)
COM1:	COM2:			(the two serial ports)
CON	KYBD			(the keyboard)
SCRN				(video display)
AUX				(the auxiliary)
CLOCK$				(the clock)

The lower nibble of the reserved variable ERDEV can also give you some useful information, mostly for file handling programs (use ERRDEV AND &HF to get these bits out). For example, if ERDEV AND &HF = 9 then the printer is also out of paper. See appendix G for these codes.

There's one other error handling function—ERL (ERror Line). If your program has line numbers, then this will give you the line number for the error.

When developing a program you may want to test how your error handler works. QB includes the statement:

ERROR errorcode number

which, when processed, makes the compiler behave as if the error described by the given error number had actually occurred. This makes it easier to develop the "trap."

Just like enabling the keyboard break option makes your standalone program a bit slower, so too does error trapping. However, this difference is rarely important enough to throw away the advantages that a well written error handler provides. On the other hand you're always better off if you don't need to activate the error trap at all. This is best done by checking the data before you use it. A robust program is best off not relying only on error trapping.

If you are confident that you will no longer need an error trap, then you may disable error trapping with the statement:

```
ON ERROR GOTO 0
```

Similarly, you can change which error trap is in effect by another ON ERROR GOTO statement. QB uses the last such statement processed to decide where to go.

6
CHAPTER
Lists, arrays and records

You've seen two methods of assigning values to variables:

1. A LET (assignment) statement;

and

2. By responding to various kinds of INPUT statements.

In both these methods, the variables are named and their values assigned at essentially the same time. The methods in this chapter are different: you set aside space in the computer's memory that can be named and use in a systematic way, at any time during the course of the program. You'll also see a more efficient method of building frequently used information inside a program. Finally, this chapter covers some of the many methods known to search and sort lists. The next chapter covers more sophisticated methods.

Getting started with lists

Suppose you are writing a program that requires one variable with the value 1, a second with the value 2, a third with the value 3, and so on, for 100 different variables. The outline for this kind of program cries out for a FOR-NEXT LOOP:

```
FOR I = 1 to 100
    Assign I to its variable
NEXT I
```

Or consider the graphing program. This program used a user-defined function to accept the numbers. Moreover, I had to limit the function to accepting numbers smaller than 23 so that the bars would fit. Suppose, for example, that you wanted to allow values as large as 46. Then you could keep the bars in proportion, and still have have them fit, if you made each bar stand for two units. In general, to allow arbitrarily large or small values: first find the largest value—once you know that value, use it to scale the bars.

If the largest value was stored in a variable named Large, multiplying its value by 23/Large shrinks it to fit into 23 units. Now multiply all the other values to be graphed by this scaling factor. Since we're assuming Large contains the largest value, all the other variables will not fit into a 23 unit space as well.

Anyway, you already know how to find the largest value on a list, and you also know how to write the procedure to graph a bar. The only thing left is to see how to temporarily store the values while you're determining the scaling factor (the largest value). For example, suppose you want to do a bar graph for a 12 month period. You would like to say something like:

```
FOR I = 1 to 12
    INPUT MI
     IF MI > Large THEN Large = MI
NEXT I
```

If this were possible, then the information just entered would still be available in the variables MI. Unfortunately, this is not quite correct. In QuickBASIC, MI is a perfectly good variable name—for a single variable. QuickBASIC cannot separate the I from the M. After you see a systematic way to name variables, it's easy to take care of these problems.

To QuickBASIC a list is just a collection of variables, each one of which is identified by two things:

(1) The name of the last

and

(2) its position on the list.

The third errand on a list that mimics a pre-printed errand form might be called ERRAND$(3). The name of this list is Errands$—notice the dollar sign ($) that indicates the variables on this list will hold words (strings). The number in the parentheses is usually called a subscript or pointer. The term subscript comes from mathematics where the item M(5) is more likely to be written M_5. The term pointer comes because the 5 "points" to the row holding the information. To QuickBASIC, M5 is the name of a single variable, but M(5) is the name of the fifth element on a list called M, and, of course, a program can use both.

Lists can't be open-ended in QuickBASIC. While the limits are quite large—you can have up to 32768 items on certain kinds of lists—you must tell the compiler how much memory to set aside for the list before you use it. Fortunately, it is never necessary to say exactly how long the list is. Just make sure that it is at least as long as the number of items that you want it to contain. The statement that gives the length of a list is called a dimension statement, and the command is DIM. For example, if you wanted to have an errand list that allowed 30 errands you would have the statement DIM ERRANDS$(30). The dimension statement must come before you refer to any elements in the list. Values can then be assigned using a FOR-NEXT loop, or, since we want to allow someone to stop before entering all 30 names, by a DO loop, as in listing 6.1:

```
' CH6\P1.BAS
' A simple list demo

CLS
DEFINT A-Z
DIM Errand$(30)
Index = 0

DO
  PRINT "You've entered";Index;"entries so far."
  INPUT "Enter the next errand - ZZZ when done"; Errand$    'A
  IF Errand$ <> "ZZZ" THEN                                  'B
    Index = Index + 1
    Errand$(Index) = Errand$
  END IF

LOOP UNTIL Index = 30 OR Errand$ = "ZZZ"

NumberOfItems = Index                                       'C

END
```

A) Notice that you set up a temporary variable Errand$. This keeps the flag off the list. It also shows that you can have a variable with the same name as a previously dimensioned list.

B) Once you know the entry is acceptable, you first move the pointer, or add one to the Index, and only then fill in the entry. Do you see why the order is important? Notice as well that this IF THEN is completely skipped when ZZZ is entered. This keeps the flag off the list. Enter a ZZZ, and we move immediately to the loop test and leave.

C) This keeps track of the number of items on the list. In a program that will manipulate this list in many ways, it's a good candidate for a shared or global variable.

However there are other ways. In QuickBASIC all lists come with a zeroth entry. In your situation, when you said DIM Errand$(30), QuickBASIC actually set aside 31 slots, the extra one being Errand$(0). This zeroth slot is useful for things like the number of significant items on the list. What I would probably say, rather than set up a new variable, is:

```
Errand$(0) = STR$(Index)
```

Now, to find the number of items, for example to set up a FOR-NEXT loop, I can convert this entry back to a number using the VAL command.

Another, perhaps even more popular, alternative is to keep a flag, like the ZZZ, as the last item on a list. This lets you use an indeterminate loop to manipulate the list. To do this, modify the previous fragment as here:

```
' CH6\P2.BAS
' A simple list demo revisited and revised

CLS
DEFINT A-Z
DIM Errand$(30)
Index = 1

DO
  PRINT "You've entered";Index -1;"entries so far."
  INPUT "Enter the next errand - ZZZ when done"; Errand$
  IF Errand$ = "ZZZ" THEN
    Errand$(Index) = "ZZZ"
  ELSE
    Errand$(Index) = Errand$
    Index = Index + 1
  END IF
LOOP UNTIL Index = 30 OR Errand$ = "ZZZ"

NumberOfItems = Index

END
```

Notice that here I've used the IF-THEN-ELSE to deal with the two possibilities—a real entry, or the flag. Notice as well that the last entry could hold the flag.

Of the two methods, I find the idea of keeping the number of items currently used in the zeroth entry, when possible, the most appealing, but this is clearly a matter of taste. I find it comforting to always know how many entries are on a list. It makes debugging easier and I, like most programmers, find FOR-NEXT loops easier to use than DO loops.

More on lists

Some people never use the zeroth entry of a list—they just find it confusing. And, if you are not going to use it in a program, it certainly wastes space. For this reason QuickBASIC, to keep compatibility with interpreted BASIC, has a command that eliminates the zeroth entry in all lists DIMensioned from that point on. It is the OPTION BASE 1 statement. After this statement is processed all new lists DIMensioned begin with the first. After OPTION BASE 1, DIM Errand$(30) sets aside 30 spots, rather than 31.

However QuickBASIC goes interpreted BASICs one step better. Suppose you wanted to write the input routine for a bar graph program for sales in the years 1981 through 1987. You could say something like:

```
DIM SalesInYear(7)
FOR I = 1 TO 7
    PRINT "Enter the sales in year"; 1980 + I
    INPUT SalesInYear(1980 + I)
NEXT I
```

However, this program requires 14 additions, two for each pass through the loop, and is more complicated to boot. This situation is so common that QuickBASIC decided to enhance the language by allowing subscript ranges. Instead of saying:

```
DIM SalesInYear(7)
```

we can now say:

```
DIM SalesInYears(1981 TO 1987)
```

The keyword TO marks the range, smaller number first, from 1981 to 1987 in this case, for this extension of the DIM command. Using this version of the DIM statement we can rewrite the fragment above as:

```
DIM SalesInYear(1981 TO 1987)
FOR I = 1981 TO 1987
    PRINT "Enter the sales in year"; I
    INPUT SalesInYear(I)
NEXT I
```

Besides being much cleaner, this new fragment runs faster. In a large program with lists with thousands of entries, the savings can be substantial.

QuickBASIC makes it easy to use short lists. If a list has no more than 11 items, you don't need a dimension statement. As soon as the compiler sees there is a subscript on a variable that has not previously been dimensioned, it assigns 11 places numbered from 0 to 10 for that item. Of course if what the compiler sees first is ITEM$(30) with no dimension statement before then you'll get an error message. For this reason, if for no other, it's a good idea to dimension all lists, whether short or long ones.

Suppose a teacher wanted to write a program to start analyzing the data from a test or group of tests. Since both the number of students and the number of tests might vary from class to class, it would be better to enter the class size and number of tests before dimensioning. This can be done by using a variable instead of a constant in the DIMension statement. Doing this is called *dynamic dimensioning*. For a single exam you could use:

```
    INPUT "How many students in this class";StuNumber
    DIM Grades(StuNumber),StuNames$(StuNumber)
```

or to allow more than one test:

```
    INPUT "How many students in this class";StuNumber
    INPUT "How many tests";NumOfTests
    DIM Grades(StuNumber,NumOfTests),StuNames$(StuNumber)
```

Now the list will hold a row for each student, and a column for each exam. Notice that in both these lists the same row number marks the name of the student, and then his or her result on the exam.

When you dynamically dimension a list or array, the compiler sets aside space at that time. If the program is very large, then it's possible that you won't have enough room. If this happens then you'll get the dreaded Out of memory error. At this point the only cure is to first refer to the following table:

List Type	Maximal Number of Entries	Requirement/ Entry
Integer	32768	2 bytes
Long Integer	16384	4 bytes
Single Precision	16384	4 bytes
Double Precision	8192	8 bytes
String	16384	4 bytes

However, the total number of characters that can occur as values of string variables in a program is the same as the size of a file you can edit at one time. Both are limited to 64K = 65536 characters.

To bulletproof a program that you think is getting close to the limits on memory, use the various forms of the FRE command:

FRE(−1)	gives the amount of room for arrays.
FRE("string")	gives the amount of room for strings (data). You can use any string expression here.

For example, suppose the average name uses 20 characters. Then you might use these commands in something like listing 6.3:

```
DO
  CLS
  INPUT "Number of names";NumOfNames
  IF NumOfNames*20 > FRE("string") THEN            'A
```

```
      PRINT "Not enough room for that many names."
      BEEP
   ELSE
      DIM Name$(NumOfNames)
      EXIT LOOP                                   'B
   END IF
LOOP
```

A) If each name takes up on average 20 characters, equals 20 bytes because one character is one byte, then this line checks that there's enough room in string space for all the names. Notice that I didn't check the amount of list space (FRE(−1)), as perhaps I should have. It's worth keeping in mind that the space set aside for a list doesn't contain the actual characters. It contains only the information on where the compiler will find them.

B) I chose to use the EXIT LOOP command rather than set up a flag variable and use a LOOP UNTIL. I think it gives a cleaner feel to the program.

As your programs grow longer, the possibility that you'll run out of space increases. Although given QuickBASIC's rather large limits, it's never all that likely. QuickBASIC allows you to reclaim the space used by a dynamically dimensioned array. This is done with the ERASE command. For example,

```
ERASE ERRAND$
```

would erase the ERRAND$ array, and free up the space it occupied.

If an array was not dimensioned dynamically, i.e., using the value of a variable, for example, DIM E(30), then the ERASE command simply resets all the entries back to zero for numeric lists, to the null string, for string lists. These kind of lists are called static lists. Using the ERASE command on a static list gives a fast method to "zero out" the entries.

On the other hand you may occasionally want to dimension an array dynamically, so that you can reclaim the space if necessary. You can do this, for example, by saying:

```
Size = 3000
DIM E(Size)
```

rather than:

```
DIM E(3000)
```

This sets up a dynamic, ERASEable, array with 3000 single-precision numeric entries. Remember all lists that do not use variables on the dimension statement are static.

Another advantage to dynamic dimensioning is that, with a little work, you can enlarge the dimensions, if you goofed. The command REDIM listname sets aside a larger space for the list given by the name. Unfortunately, it also throws away all the information that was there to begin with. To get around this you have to:

Set up a temporary list for the old information.
Copy the information to the temporary list.
REDIM the original list.
Now recopy the information from the temporary list back again.

Obviously, this is a lot of bother, and can take a fair amount of time, but dynamic dimensioning does give you this option. This is often needed in error traps—see chapter 10.

The type of a list is usually given by an explicit type identifier. DIM E%(3000) is an integer list, DIM E#(3000) a list for double-precision numbers. The default, as in the list E above, is, just like for variables, single precision. However, also just like for variables, you can change the default by a DEFINT, DEFDBL etc. For example, if a DEFINT A-Z is in effect, then DIM E(3000) sets up a list of integers.

I said above that QuickBASIC assumes that arrays dimensioned with constants are static. You can change this, if you want to, by using a metacommand. A metacommand is a statement that affects how the compiler treats your program. The one you need is $DYNAMIC. All metacommands must begin with REM or the apostrophe. They are not executable statements. Put:

```
REM $DYNAMIC     or     ' $DYNAMIC
```

as the first line of a program, and the compiler assumes that all lists are dynamic; actually, all but those implicitly defined without DIMensioning. This continues unless the compiler later encounters another metacommand: $STATIC. This one restores the normal way QuickBASIC handles lists. So if you place the $DYNAMIC metacommand before any lists are dimensioned, then all the space used for a list can be reclaimed, by a judicious use of the ERASE command.

Arrays

You can also have variables with more than one subscript, called arrays. Just as lists of data lead to a single subscript, tables of data lead to double subscripts. For example, suppose you wanted to store a multiplication table in memory, as a table. You could do this as:

```
DIM MultTable(12,12)
    FOR I = 1 TO 12
        FOR J = 1 TO 12
```

```
            MultTable(I,J) = I*J
          NEXT J
        NEXT I
```

To compute the number of items in an array, you multiply the number of entries. The dimension statement here sets aside either:

144 (12*12) entries if an OPTION BASE 1 has been previously processed, or

169 (13*13) entries if an OPTION BASE 1 has not been processed.

So, in this example, there are either 12 rows and 12 columns or 13 rows and 13 columns. In general, the number of entries in an array is either the product of the numbers, or the product of one more than the individual numbers used in the dimension statement.

The convention is to refer to the first entry as giving the number of rows and the second the number of columns. So, following this convention, I would describe this fragment as filling an entire row, column by column before moving to the next row.

As you can see in arrays the extra space taken up by the zeroth row and zeroth column can dramatically increase the space requirements for your arrays. The total number of entries cannot exceed the numbers given on the previous table. For this reason, QuickBASIC's range feature is even more welcome. The statement:

```
DIM Salary(1 TO 50, 1980 TO 1987)
```

sets aside 50 rows, numbered 1 through 50, and 8 columns numbered 1980 through 1987. So this Salary array has 400 entries.

QuickBASIC allows you up to eight dimensions, you could, in theory:

```
DIM LargeArray%(2,2,2,2,2,2,2,2)
```

which would set aside either 2^8 = 256 or 3^8 = 6561 entries, depending on whether an OPTION BASE 1 statement has been processed. But, I've never used more than 4 dimensions in a program—even a three dimensional array is uncommon.

For a more serious example of a program using arrays, consider the following program which constructs a magic square. A magic square is one where all the rows, columns and long diagonals add up to the same number. They were once thought to have magical properties. The most famous one is probably:

```
16  3  2 13
 5 10 11  8
 9  6  7 12
 4 15 14  1
```

which occurred in Albert Dürer 's famous print called "Melancholy," engraved in 1514.

Many people have devised rules for constructing magic squares. The one I'll use is called Loubère's rule, and only works for odd order magic squares—those with an odd number of rows and columns. Here's the method:

1. Place a one in the center of the first row.
2. The numbers now go into the square in order, by moving up on the diagonal to the right.

   Of course, you're immediately met with the problem of where to put the 2. If you've placed a 1 in the top row, going up takes you off the square. The solution is:
3. If you go off the top, wrap around to the corresponding place in the bottom row.

   On the other hand, going to the right eventually drops you off the side. In this case, use:
4. If you go off to the right hand, wrap around to the left column.

Finally, if a square is already filled, or the upper right hand corner is reached, move down one row and continue applying these rules.

Here's a 5×5 magic square constructed with this rule:

```
17 24  1  8 15
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9
```

The following program is a bit more clumsy than necessary. Once I show you how to use arrays with procedures, it can be cleaned up considerably. Here's the program:

```
DECLARE FUNCTION NxtRow% (X%)
DECLARE FUNCTION NxtCol% (X%)

'CH6\P4.BAS
' Magic squares by Loubere's rule

DEFINT A-Z

DO
 CLS                                              'A
 PRINT "Number of dimensions - must be odd - maximum of 15";
 INPUT NumOfDim
LOOP UNTIL NumOfDim MOD 2 = 1 AND NumOfDim < 16

Row = 0                                           'B
Col = NumOfDim \ 2
Limit = NumOfDim - 1
DIM Magic(Limit, Limit)
SIZE = NumOfDim * NumOfDim
Magic(Row, Col) = 1
```

```
FOR I = 2 TO SIZE

   IF Row = 0 AND Col = Limit THEN                        'D
     NewRow = 1
     NewCol = Col
   ELSE
     NewRow = Row - 1
     NewCol = Col + 1
     NewRow = NxtRow(NewRow)                             'E
     NewCol = NxtCol(NewCol)
   END IF

 ' find empty slot
    DO UNTIL Magic(NewRow, NewCol) = 0                   'F
         NewRow = Row + 1
         NewCol = Col
    LOOP

      Row = NewRow
      Col = NewCol
      Magic(Row, Col) = I

NEXT I

' print completed square
CLS                                                       'G
StartCol = 40 - (.5 * 5 * NumOfDim)
StartRow = 13 - (.5 * NumOfDim)
LOCATE StartRow, StartCol

FOR I = 0 TO Limit
  FOR J = 0 TO Limit
     PRINT USING "#####"; Magic(I, J);
  NEXT J

  StartRow = StartRow + 1
  LOCATE StartRow, StartCol
NEXT I

END

FUNCTION NxtCol (X)

SHARED NumOfDim, Limit

IF X > Limit THEN
  NxtCol = X - NumOfDim
ELSE
 NxtCol = X
END IF
END FUNCTION

FUNCTION NxtRow (X)

SHARED NumOfDim
```

```
IF X < 0 THEN
  NxtRow = X + NumOfDim
ELSE
 NxtRow = X
END IF

END FUNCTION
```

A) This loop continues until the user enters an odd number; that's what NumOfDim MOD 2 = 1 means.
B) This block initializes the various blocks. For example, the number of entries in a 7 by 7 magic square is 7*7 = 49.
C) You could, of course, use a DO loop.
D) This corresponds to the special case of the right upper corner.
E) This is the first rule, up and to the right. Of course, this row may be off the square and the functions FNNxtRow and FNNxtCol take care of this (see (H) and (I)).
F) This loop stops when we get to an unoccupied square.
G) This prints the square.
H, I) You wrap around by adding the number of rows, to move to the bottom row, and subtracting the number of columns, to move to the left most column.

As I said above, bear with me, once you see how to combine arrays with procedures, this program can be made much cleaner.

READing DATA

Getting information, or data, into a computer is not one of life's great joys. In the good old days, say 15 or 20 years ago, hundreds of people sat in a room, each one facing a large noisy machine that punched holes in cards. You took those cards, and then another noisy machine read the punch cards and sent the information into the computer, to be run a few hours or days later. With microcomputers programmers have thankfully put the punch card, or batch job, era behind them. Nonetheless, typing in hundreds of LET or INPUT statements is not my idea of fun. If the data doesn't change much it's silly to have to re-enter it each time the program is run. Using INPUT statements means you'll have to retype the information each time. Lots of assignment (LET) statements can make even a simple program too long.

The way around this is to build the data into the program. The statement that does this is called, naturally enough, the DATA statement. To grab information from a DATA list, use the READ statement. For example, here's a fragment that starts to print the old song "On the Twelfth Day of Christmas":

```
' Using the READ DATA combination
DIM Days$(12),Gifts$(12)
```

```
        FOR I = 1 TO 12
            READ Days$(I),Gifts$(I)
        NEXT I
    DATA "first", "a partridge in a pear tree"
    DATA "second", "two turtle doves and"
    DATA "third", "three french hens"
    .
    .
    .
```

As soon as QuickBASIC sees a READ statement, it looks for a DATA statement with something to READ. The READ command takes input from the DATA statements in order, much like an INPUT statement works. In this program, on the first pass through the loop, QuickBASIC reads the word first and assigns it to Days$(1). Because of the comma it stops before the phrase. Similarly, it reads the phrase "a partridge in a pear tree" and assigns it to Gifts$(1). On the second pass it fills up Days$(2) and Gifts$(2), and so on. The nested loop, combined with the READ DATA combination, replaces 24 assignment statements.

To actually display the song, finish creating the DATA statements and add the following:

```
FOR I = 1 TO 12
        FOR J = I TO 1 STEP -1
        PRINT "On the ";Days$(I);" day of Christmas, my true love gave"
        PRINT "   to me ";Gifts$(J)
    NEXT J
NEXT I
```

Here I gives the day and J runs backwards, starting from the correct day—as it does in the song.

In a DATA statement each entry is separated from the next by a comma, and no comma follows the last entry. You can put as many items in a single DATA statement as comfortably fit on a line. It's a good idea to place quotes around strings as I did, although this is only really necessary if the string has a comma or carriage return inside of it, because QuickBASIC uses these as separators. If you put in more DATA than a program needs, then the compiler reads only what it needs. On the other hand if you ask for more data than is contained in the DATA statements, you'll get an Out of DATA error message.

Essentially what QuickBASIC does is keep an invisible pointer at the last item it READ. Each time it reads another item, it moves the pointer forward. The DATA statements are read in the order they appear in the source code. All items in the first DATA statement are read, then what's contained in the next data statement, and so on.

Occasionally, you need to reset this pointer yourself. This is done with the RESTORE command. A bare RESTORE command sets the pointer back to the beginning of the very first DATA statement, and all the information in all the DATA statements is available again. This is especially useful when you want to dynamically dimension a list according to the number of items in a DATA list. Read through the list twice. On the first pass through the list, have the program count the number of items until it encounters a flag, something like 1E32 or "ZZZZ." Now dynamically dimension the list using this count. Next, use the RESTORE command to reset the pointer back to start. Finally, have the program re-READ the DATA.

The RESTORE command is useful, but sometimes it's overkill to go all the way back to the first DATA statement. You may not want to go back to square one, you want to go back to a specific place, a specific DATA statement. To do this you need to use a label. In earlier BASICs line numbers were the only way to label lines. QuickBASIC allows you to use both line numbers and a mnemonic description. This is done by placing the label followed by a colon (:) on a blank line immediately preceding what you want to label. For example:

```
Titles:
    DATA "Foundation", "I Robot", "Nightfall"
    DATA "Gulf", "Misfit", "Citizen of the Galaxy"
    DATA "Flandry", "Vault of Ages"
Authors:
    DATA "Asimov", "Heinlein", "Anderson"
```

The statement RESTORE Authors sees the pointer at the first item in the DATA list, following the Authors label, to Asimov in this case. The statement RESTORE Titles sets the pointer at the string "Foundation."

Using lists and arrays with procedures

QuickBASIC has as extraordinary facility to use lists and arrays in procedures and functions. Unlike languages like Pascal, as you'll soon see, it's easy to send any size list or array to a procedure. One obvious way, of course, to do this is to make the list or array a SHARED variable.

Suppose you want to make both a list called ExOfList( ), and an array called ExOfArray, shared variables in a function. The syntax for doing this is:

```
FUNCTION Example
SHARED ExOfList( ),ExOfArray( )
```

where the empty parentheses tell the compiler that the variable stands for a whole list, or an array. Similarly:

```
SUB Example
SHARED ExOfList( ),ExOfArray( )
```

You can also create a list or array that is local, or STATIC, within a procedure by DIMensioning it within the function or procedure:

```
FUNCTION NextExample
SHARED ExOfList( ),ExOfArray( )
' LOCAL lists will be LocalList( ),LocalArray( )
DIM LocalList(10 TO 20),LocalArray(4,3)
.
```

or

```
SUB SimpleExample
SHARED ExOfList( ),ExOfArray( )
'LOCAL lists will be LocalList( ),LocalArray( )
DIM LocalList(10 TO 20), LocalArray(4,3)
.
.
```

One way to turn this to your advantage is to send a procedure a parameter that controls the dimension of the list or array. For example:

```
CALL LessSimpleExample(SizeOfList,NumOfRows,NumOfCols)
'LOCAL lists will be LocalList( ),LocalArray( )
DIM LocalList(SizeOfList),LocalArray(NumOfRows,NumOfCols)
.
.
.
```

Space for a local list or array is automatically reclaimed when the procedure or function call is finished. Space for a STATIC list, or array, is not. This lets you CALL a procedure or function that uses a local list or array repeatedly without quickly running out of space, although you can only use it from within the procedure or function—it's invisible to the rest of the program.

Although you can begin to take advantage of QuickBASIC's power by judicious use of local, SHARED, and STATIC lists, to take full advantage of QB's power, you'll want to send a list or array as a parameter to the procedure. The advantages of doing this are the same as why one usually prefers parameters to shared variables—they increase flexibility. To send an array parameter to a procedure or function, put the name of the array followed by a ( ) in the parameter list. For example, let's assume that Array$ is a two dimensional string array, and BigArray% a three dimension array of integers. Then:

```
SUB Example(List#( ),Array$( ),BigArray%( ),X%)
```

would allow this Example procedure to use and change a list of double-precision variables, an array of strings, a three dimensional array of integers and a final integer variable. Note that just as with variables, list and array parameters are placeholders—they have no independent existence. To call the procedure one might have a fragment like this:

```
    DIM PopChange#(50),CityState$(3,10),TotalPop%(2,2,2)
```

Now:

```
    CALL Example(PopChange#( ),CityState$( ),TotalPop%( ),X1#)
```

would call this procedure by sending it the current location (pass by reference) of the three arrays and the integer variable. And, just as before, since the compiler knows where the variable, list, or array is located, it can change the contents.

Suppose you wanted to write a function procedure that would take a list of numbers and return the maximum entry. Since you may want to do this for many different lists, you decide to write a procedure that follows this outline:

```
FUNCTION FindMaximum(List( ),Max)—
Start at the top of the list.
Each time an entry is bigger than the current Max, swap it.
Until you finish the list.
```

This kind of outline obviously calls for a FOR-NEXT loop. But the problem with translating this outline to a program is: how do you know where the list starts or ends? You could arrange for every list to have a flag at the end, but then you would have trouble combining this with QuickBASIC's range feature. Or, you could use the trick of reserving one entry in the list for the number of items in the list.

QuickBASIC makes this process easier by the commands LBOUND and UBOUND, that are not part of interpreted BASIC. LBOUND gives the lowest possible index, and UBOUND the highest in a list. For example, you can easily translate the outline to listing 6.5:

```
SUB FindMax(A(),Max)

' LOCAL variables Start%,Finish%,I

Start% = LBOUND(A)
Finish% = UBOUND(A)

Max = A(Start%)

FOR I = Start%  TO Finish%
  IF A(I) > Max THEN Max = A(I)
NEXT I

END SUB
```

When this procedure is finished, the new value of Max would be the largest entry on the list.

In general, the command LBOUND(NameOfArray,I) gives the lower bound for the ith dimension. For a list, the 1 is optional, as in the previous example. So:

```
DIM Test%(1 TO 5,6 TO 10,7 TO 12)
PRINT LBOUND(Test%,2)
```

gives a 6, and

```
PRINT UBOUND(Test%(3))
```

gives a 12.

Here's another example. Suppose you wanted to write a general procedure to copy one two dimensional string array to another. The LBOUND and UBOUND commands allow you to copy list or arrays with different ranges, providing the total number of rows and columns are the same. Subtract the LBOUND from the UBOUND of both for each dimension and see if they match.

It's hard to stress enough the flexibility that QuickBASIC's method for handling lists and arrays within procedures gives, especially when combined with the LBOUND and UBOUND commands. For example, some of you may have learned about matrices in your math or engineering courses. It is close to impossible to write a general matrix package in standard Pascal, but it's almost trivial in QuickBASIC.

I should mention that you cannot pass an array by value to a procedure; you can only pass by value a specific entry in the array. To do this, give the entry. For example:

```
CALL Example(A(),B(10))
```

would send a procedure a list named A, and the value of the tenth entry in another list named B. You may be wondering why you can't pass lists and arrays by value. It's because, as I mentioned before, when QuickBASIC passes by value it makes a local copy of the variable. As you can well imagine, making local copies of a list with 200,000 entries would quickly run out of memory.

One last point; the LBOUND and UBOUND are not a cure-all. If part of the list or array hasn't yet been filled, then they may not help. This is why the trick mentioned in the first section is sometimes useful. It was invented for earlier BASICs that didn't have UBOUND and LBOUND.

Searching

Suppose a long list of names is stored in the computer's memory. Now you want to find out if a certain name is on the list. This can be done easily: just write a program to compare the name you want with all the names on the list. Because a compiler like QuickBASIC, works fairly rapidly, this method is quite effective for short lists. However, if the list had 5000 names and was already in alphabetical order, then this would be a silly waster of time. If you are looking in a telephone book for a name beginning with K, you don't start at page 1. You split the book roughly in half and proceed from there. When the information on the list you're searching is already ordered you can speed things up by an extension of this method. Each time the program will look at a list that is only half the size of the previous list. This speeds things up almost beyond belief. Here's an outline for a program to search through a list that is already in alphabetical order.

1. Divide the list in half.
2. Has it gone too far? Is the entry at the halfway mark before or after the name it's looking for?
3. If it has gone too far, look at the first half—if not look at the second half.
4. Go back to step 1 as long as there are names left to look at.

Suppose your list had 5000 names. Then, after doing step 4, you go back to step 1 with a list of 2500 names. Doing it again you have only 1250 names, the third time only 625, and so on. By the twelfth time there would be only two names to look for. In computerese what I just described is called doing a *binary search*.

An extraordinary feature of binary search is that this method is almost as fast for large lists as for small. In particular, suppose you are searching through the N.Y.C. telephone directory, with roughly 10,000,000 entries, and you had to find out if someone's name is there. Just by following this outline, and not doing any estimating of where the letters are, you would find out if the name was in a list of names in no more than 25 applications of step 4.

The procedure is a bit tricky, so it's worth spending a bit of time on. Here is a first attempt:

```
DECLARE SUB BinarySearch (A$(), Target$)

SUB BinarySearch (X$(), Target$)

' LOCAL variables are: Low, High, Middle
SHARED TargetPosition

TargetPosition = 0                              'A
Low = LBOUND(X$)
High = UBOUND(X$)                               'B

DO
  Middle = (Low + High) \ 2                     'C

  SELECT CASE X$(Middle)                        'D

    CASE IS = Target$
      TargetPosition = Middle
    CASE IS > Target$
      High = Middle - 1                          'E
    CASE IS < Target$
    Low = Middle + 1

  END SELECT

LOOP UNTIL TargetPosition <> 0                    'F
PRINT TargetPosition
END SUB
```

A) This initializes the shared variable TargetPosition. At the end of the procedure it will contain the position of the target. You also could set up another parameter for this information, or, an even better idea would be to turn the whole thing into a FUNCTION whose value was the location of the target.
B) I'm using the UBOUND/LBOUND method of finding the limits. The method described earlier to store the number of entries in the list as the zeroth entry is not needed here, since I'm assuming the whole list is ordered.
C) I'm using the integer division operator—list indices are always integral.
D) Because there are three possibilities in the search it's a perfect candidate for the SELECT CASE command. You can also use the IF/THEN/ELSEIF, of course.
E) If the entry in the middle position is too large, then you know that you should look at the first half of the list. Because it can't be the middle entry—you've eliminated that in the first CASE—you can move the high index down by one. A similar situation holds in the next case.

Now you get to the problem in this preliminary version of binary search. The loops stops only if TargetPosition has a nonzero value, or, in other words, if you have found the entry. But, suppose the entry wasn't on the list? The loop never stops—I've set up an infinite loop.

How can I fix this module, so that it stops when there are no more entries left to check? As usual a concrete example that you can play computer with helps. Suppose you are down to a list that consists of two names, say, in the 42nd and 43rd positions. The 42nd entry is too small, and the 43rd entry too large. What does this procedure do? The first time you're in this situation the value of middle is set to $(42+43)\setminus 2 = 42$. Because you're assuming the value in the 42nd position is too small, the value of low is set to one more than middle, i.e., 43. The value of low and high are now the same. What happens next? Because both low and high are the same, the value of middle is also the same. Now the entry in the middle position is too large, so the value of high shrinks by one. It's now 42—less than the value of low. This gives you one way to end the loop. Change it to read:

```
LOOP UNTIL TargetPosition < > 0 OR High < Low
```

There's another way to write this loop that some people find easier to understand: it depends on realizing that something special happens for small lists, say when the difference between high and low is one. Arrange to leave the loop when the list has size one and add a few lines to take care of this special case:

```
IF (High - Low) < = 1 THEN
    IF A$(High) = Target$ THEN TargetPosition = High
ELSE
    IF A$(Low) = Target$ Then TargetPosition = Low
END IF
```

Notice that both these possibilities take care of the case when there is only one or even no entries on the list. As I mentioned in the last chapter, it's the boundary cases that often cause the most subtle bugs.

While we're on the subject of bugs, how do you write a test module for a binary search module? Obviously you need a long, ordered list. One way to do this is to have the list consist of all possible two letter strings:

AA,AB,AC, ... BA,BB, . . . ZZ

There are 26*26 = 676 two letter combinations—using three letter combinations allows a list of 26^3 = 17576 entries. To create this list you can try this:

```
DEFINT A-Z

DIM A$(1 TO 676)
Index = 1

FOR I = 65 TO 90
 FOR J = 65 TO 90
  A$(Index) = CHR$(I) + CHR$(J)
  Index = Index + 1
 NEXT J
NEXT I
```

Now you can try the binary search module with various possibilities— a two letter string which is on the list, and one which is not—or test it with the first entry and the last.

Sorting

Programmers like ordered lists, just as people prefer alphabetized lists like dictionaries and telephone books, because techniques like binary search techniques work so fast. Rearranging—sorting—ordering data is one of the most common demands placed on a computer. Unfortunately, sorting is also one of the most time-consuming tasks a computer can be asked to do. Because of this computer scientists have developed literally hundreds of different ways to sort lists, and it's impossible to say which is best in all possible circumstances. In this section you'll see four methods. The first two are only useful for short lists, the third is often the method of choice—even for lists having thousands of entries. The last one is called bubble sort, and I give it only to condemn it. Although it seems to be the one that is most commonly given in elementary books, it has few redeeming features. And is best forgotten.

The next chapter discusses three more sorting methods. Those three are usually better than even the fastest sort presented in this section—unfortunately they are much more difficult to program.

When you sit down to write a program, it's always a good idea to ask yourself if there's anything you do in "real life" that's analogous to what you want the computer to do. For sorting lists, what comes to my mind is ordering playing cards hands. As far as I can tell, there are two types of people: those who pick all the cards at once, and sort their hands by first finding the smallest card, then the next smallest, and so on; and those who pick up one card at a time, scan what they have already sorted, and then immediately place the new card in the correct place. For what it's worth, computer scientists have proved that these two methods take roughly the same amount of time, with the second method usually being a tiny bit faster.

Each of these methods translates into a way to sort lists. The first is usually called ripple sort. An outline for it is:

Start with the first entry.

Look at the remaining entries one by one. Whenever you find a smaller entry, swap it with the first.

Now shrink the list—start with the second entry, and look at the remaining entries (3, 4 . . .), etc. Notice that if, say, the list had 50 entries then we only have to do the etc. 48 times. This is because, by the time this procedure works its way to the last entry, enough switching has happened so that it has to be the largest entry.

This is the procedure:

```
DECLARE SUB RippleSort (A$())

DEFINT A-Z

SUB RippleSort (A$())

'LOCAL Variables are:NumOfEntries%,NumOfTimes%,I%,J%

NumOfEntries% = UBOUND(A$)                    'A
NumOfTimes% = NumOfEntries% - 1

FOR I% = 1 TO NumOfTimes%
  FOR J% = I% + 1 TO NumOfEntries%
     IF A$(J%) < A$(I%) THEN SWAP A$(J%), A$(I%)
  NEXT J%
NEXT I%

END SUB
```

A) I'm assuming the list starts from 0. An even more elegant idea is to make this procedure depend on two more parameters, say low and high, and use these to establish the bounds on the loops.

How do you write a module to test a sort? Well, you need a way of creating random lists of strings. Here is one way:

```
DEFINT A-Z

DIM B$(100)
RANDOMIZE TIMER

FOR I = 1 TO 100
  RndInt = INT(26*RND(1)) + 1
  B$(I) = CHR$(RndInt+ 65)
NEXT I
```

Now

```
    CALL RippleSort(B$( ))
```

Now print out the list to make sure it's been sorted:

```
    FOR I = 1 TO 100
         PRINT B$(I)
    NEXT I
```

The second method, usually called insertion sort, is no harder to program. In this sort, at every stage you'll have an already sorted, smaller list. Look through the list, from the first entry to the last, until you find something smaller than the new entry. Unfortunately, unlike the case with playing cards, you have to move all the entries down by one to make room for the new entry. This leads to a slight tweak that can improve the performance considerably, and is easier to program. Instead of moving forward, move backwards. Now each time the comparison fails, move the old entry down by one. If you do this, then you'll be moving a hole along with you as you move through the list. When the comparison finally fails, you drop the new entry into the hole.
This is the procedure:

```
DECLARE SUB RippleSort (A$())

DEFINT A-Z

SUB Insertion Sort (A$())

'LOCAL Variables are: NumOfEntries%,I%,J%, Temp$

NumOfEntries% = UBOUND(A$)           'A

IF NumOfEntries% <= 1 THEN EXIT SUB    'B
```

```
FOR I% = 2 TO NumOfEntries%
  Temp$ = A$(I%)

    FOR J% = I% - 1 TO 1 STEP -1
      IF Temp$ >= A$(J%) THEN EXIT FOR        'C
      A$(J% + 1) = A$(J%)
    NEXT J%

A$(J% + 1) = Temp$                            'D

NEXT I%

END SUB
```

A) As before, you might want to make this procedure depend on a low and high parameter.
B) It's easier to eliminate this special case of the list having one entry here.
C) This loop moves entries forward until conditions are ripe for the EXIT FOR. This is when you have located the position of the hole—in preparation for:
D) This statement fills the hole.

Because these methods follow the playing card analogy closely, they were not hard to program. Moreover, for small lists they are reasonably fast. Sorting 100 strings, using the insertion sort or ripple sort, takes about 2.5 seconds on a basic PC. Unfortunately, sorting 200 entries takes about 10 seconds. Both these types of sorts have the unfortunate property that doubling the list quadruples the time. To sort a list of 3200 names, by no means a very large list, would take about 3/4 of an hour. So you can see that these are not the methods to use for lists much longer than 100 or so entries—you need to turn to a faster method. Now you can see why binary search is so nice—doubling the list only adds one step.

The first of these faster sorts that I want to show you was discovered by Donald Shell around 30 years ago. It's unusual because, while the procedure is simple, and short, understanding what makes it work is not. This is partially because there is nothing that you do in real life that's analogous to Shell sort, and partially because it's a really neat idea. Another problem is that, even after you understand why it works, it's unclear why it's so much faster than the previous two methods.

To understand Shell sort, you should ask yourself what are the advantages and disadvantages of the two previous sorting methods. One obvious disadvantage of ripple sort is that, most of the time, the comparisons in the various loops are wasted. The disadvantage of insertion sort is, that most of the time it moves objects inefficiently. Even when the list is mostly sorted, you still have to move the entries one by one to make the hole. The big advantage of ripple sort is that it moves objects efficiently. Once the smallest object gets to the top, it stays there.

In a sense then, insertion and ripple sorts are opposites. Shell decided to improve insertion sort by moving the keys long distances—as is done in ripple sort. Consider the following list of numbers to sort:

57, 3, 12, 9, 1, 7, 8, 2, 5, 4, 97, 6

Suppose instead of comparing the first entry to the second, you compare it with the seventh; instead of comparing the second with the third, you compare it with the eighth. In short, cut up the list into six different lists. Now use an insertion sort on these six small lists. After this you have six lists, each of which is sorted, while the whole list is likely to still not be. Merge the smaller lists and break it up into three new lists, first with the fourth, sixth, etc; second with the fifth, eighth, and so on. Do an insertion sort on these three smaller lists, and merge again. Now the resulting list is very close to being sorted. A final sort finishes up the process; insertion sort is very efficient when it doesn't have much work to do.

If the numbers are already sorted in a list, then you never have to break up the list into smaller lists. Instead you shift your point of view by concentrating on the different sub-lists. Also, because on the earlier passes, the entries moved fairly long distances, when you're down to the final step not many more moves are needed.

Here is a version of Shell sort:

```
DEFINT A-Z
SUB ShellSort (A$())

' LOCAL variables are NumOfEntries%,Increm%, J%, Temp$

NumOfEntries% = UBOUND(A$)
Increm% = NumOfEntries% \ 2

DO UNTIL Increm% < 1                           'A

  FOR I% = Increm% + 1 TO NumOfEntries%

    Temp$ = A$(I%)

    FOR J% = I% - Increm% TO 1 STEP -Increm%    'B
      IF Temp$ >= A$(J%) THEN EXIT FOR
      A$(J% + Increm%) = A$(J%)
    NEXT J%

   A$(J% + Increm%) = Temp$

  NEXT I%

Increm% = Increm% \ 2

LOOP

END SUB
```

A) This gives you the way of dividing the lists into smaller lists.
B) This does the insertion sort on the smaller lists. Since each entry on the smaller list differs from the next by the number given in A, the STEP command gives you a way of working with the smaller lists.

What's amazing about Shell sort is that it's so much faster than ripple or insertion sort. Although, surprisingly enough, nobody yet knows how much faster it will be in general. An easy way to get a Ph.D would be to fully analyze Shell sort. In any case to sort a list of 3600 names will take only about 95 seconds using Shell sort on a basic PC.

The speed of Shell sort depends somewhat on the numbers you use to split the list into smaller ones. These are usually called the increments, the 6, 3 and 1 that I used in the example above, and they should be chosen with care. Because the increments get smaller on each pass, Shell sort is sometimes known as "diminishing increment sort." The ones I used above, half of the current size of the list, are Shell's original choice. Today we know that you can obtain slightly better results with other increments. One of the simpler choices that gives slightly better results[1] is:

. . . 9841, 3280, 1093, 364, 121, 40, 13, 4, 1

where each number is arrived at by multiplying the preceding number by 3 and adding 1. You start with the largest increment that's smaller than the size of your list. So a list with 10,000 entries would start with an increment of 9841. In any case, no one yet knows the best choice of increments—try other sequences, and see if you get better results.

How do you write a realistic test module for a sorting routine? Obviously, you want to create a long list of random strings. This is not very difficult to do but it sometimes can take longer than the sort! For a list of random four letter strings for Shell sort:

```
DIM Test$(1 TO 3600)
    FOR I% = 1 TO 3600
    FOR J% = 1 TO 4
        Cnumber = INT(26*RND(1))
        TEST$(I%) = TEST$(i%) + CHR$(CNumber + 65)
    NEXT J%
NEXT I%
```

and CALL the various sort routines. (Actually, don't call anything but Shell sort

[1]The best series of increments I've heard about is due to Robert Sedgewick of Princeton. It's 1, 8, 23, 77, 281, 1073, 4193, 16577, or $4^{j+1} + 3*2^{j+1}$, but for small lists it's not worth the extra trouble.

or the routines from the next chapter . . .) Finally, add a routine to print out the transformed list. If the result is an ordered list, you might want to add a routine to time the various sorts. If you devise your own sort, or want to to test these, I suggest also testing the sorts on the two boundary cases. For a sorting routine these are usually thought of as, when the list is either already ordered or completely in reverse order.

Finally, I want to end this section by saying something about the "bubble sort" that seems to be so prevalent. The idea of the bubble sort is that you constantly compare an entry with the one below it. This way the smallest one "bubbles" to the top. The code for this is almost trivial:

```
FOR I = 2 TO N
    FOR J = N TO I STEP -1
        IF A$(J-1) > A$(J) THEN SWAP A$(J-1), A$(J)
    NEXT J
NEXT I
```

The problem is that it is well known that bubble sort is the slowest sort of all. So, since it has no redeeming virtues that I'm aware of, it should be replaced, at the very least, by ripple sort (which is just as easy to program) for small lists, and one of the faster sorts, such as Shell sort, for longer lists.

Case study: the eight queen problem

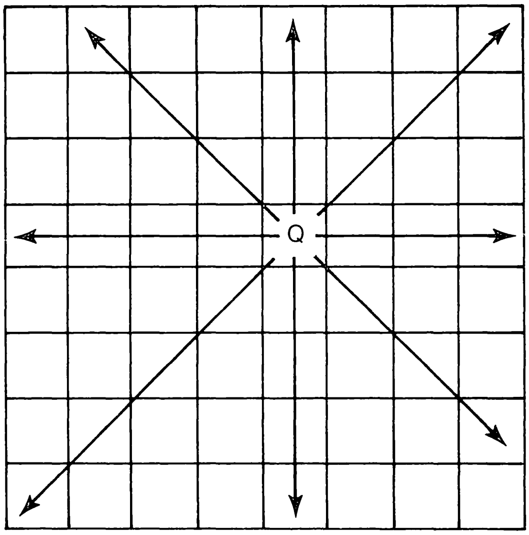
As a final example program, consider how you would represent a chess board and the squares that a queen could control from a given square as entries in an array. The chessboard might be an 8×8 array of integers.

```
DIM ChessBoard%(8,8)
```

A queen controls all the squares on the two diagonals, as well as the horizontal and vertical directions. Suppose the queen was placed as in Fig. 6-1. In this representation this would mean you might place a non-zero integer, say an 81, because CHR$(81) = "Q," in the 5,4 position. It controls all squares with either a 5 in the first position (same row), or a 4 in the second position (same column). What about the diagonal squares it controls? Looking at Fig. 6-1 you can see that the diagonal squares are:

```
8,1  7,2  6,3  4,5  3,6  2,7  1,8
8,7  7,6  6,5       4,3  3,2  2,1
```

Essentially, you can both add and subtract one from the original square, until you get to the end of the board—either a one or an eight. Another way to think about it is that on the diagonals, either the sum of the numbers is 9 (5+4), or the difference is 1 (5−4).



6-1 A queen controlling part of a board.

The point of discussing this is that I want to show you a new method[2] for solving the classic "8 queens problem." This asks: how can you place 8 queens on a chessboard so that no queen can take another. It's usually solved by a method called backtracking, that you'll see in the next chapter. Here, though, I want to present Undercoffer's method that uses random numbers. His method is:

1. Choose a square at random and put a queen on it.
2. X out all the squares controlled by the queen from step 1.
3. If there are any empty spaces left, i.e., the number Xed out is less than 64, repeat step 1.
4. If you have used up 8 queens, print out a solution—otherwise begin again from step 1.

Step 1 is easy: use the random number generator and scale the results to get an integer between 1 and 8. For step 2 you can X-out squares by starting with an array of zeros and replacing them by a non-zero integer. Step 3 means keeping track of the number of queens and the total number of squares. Finally, step 4 means ERASEing the array and starting over again.
This is the program:

```
' CH6/P7.BAS
' 8 Queens via random walk

DECLARE SUB FillBoard (X%, Y%)
DECLARE FUNCTION RandomInt% ()
```

[2]Due to Kurt Undercoffer J. of Pascal, Ada & Modula-2, Vol. 6 #2, 1987 pp. 45-50.

```
CLS
DEFINT A-Z
DIM Chessboard(1 TO 8,1 TO  8)               ' or OPTION BASE 1

CONST Queen = 81                                ' CHR$(82) = "Q"
CONST Star = 42                                 ' CHR$(42) = "*"
D = 1

RANDOMIZE TIMER

DO
   ERASE Chessboard                             'A
   SqCnt = 0
   QnCnt = 0

   DO WHILE SqCnt < 64                          'B
       Row = RandomInt: Cl = RandomInt
       IF Chessboard(Row, Cl) = 0 THEN CALL FillBoard(Row, Cl)
   LOOP

   LOCATE 25, 1: PRINT "Finished try#"; D;
   D = D + 1

 LOOP UNTIL QnCnt = 8

' Print the chessboard

LOCATE 25, 1
BEEP
PRINT SPACE$(30); "Success on try#"; D - 1;


 FOR I = 1 TO 8
   LOCATE (I + 7), 20
   FOR J = 1 TO 8
     PRINT CHR$(Chessboard(I, J)); SPC(4);
   NEXT J
  PRINT
NEXT I

END


SUB FillBoard (X, Y)                            'C

SHARED Chessboard(), SqCnt, QnCnt

' Local variables are:Rw,Cl,I,J

FOR I = -1 TO 1
  FOR J = -1 TO 1

Rw = X
Cl = Y
```

```
  IF I <> 0 OR J <> 0 THEN

   DO WHILE Rw > 0 AND Rw < 9 AND Cl > 0 AND Cl < 9
     IF Chessboard(Rw, Cl) = 0 THEN
       Chessboard(Rw, Cl) = Star
       SqCnt = SqCnt + 1
     END IF

     Rw = Rw + I
     Cl = Cl + J
   LOOP
  END IF

  NEXT J
 NEXT I

Chessboard(X, Y) = Queen
QnCnt = QnCnt + 1

END SUB

FUNCTION RandomInt
   RandomInt = INT(8 * RND(1)) + 1
END FUNCTION
```

A) This large loop cycles until 8 queens are placed. It starts by giving you a fresh canvas for each attempt via the ERASE command and resetting the count of the queens (QnCnt) and the squares (SqCnt).
B) This inner loop tries to place a single queen. First, it gets a random position on the board, and places a queen there. If that square is not yet under the control of a previously placed queen (i.e., its value is 0), then the loop calls the FillBoard procedure (see C)).
C) This procedure does the marking off of the squares. The work is done in the nested FOR-NEXT loops. Starting from the position of the queen, given by the parameters X and Y, it moves backwards and forwards until it runs off board—this is the purpose of the DO-WHILE loop. The IF/THEN prevents writing over the queen's current position.

Records

Remember one of the first examples in this chapter, the teacher and the grades? I said:

```
DIM Grades(StuNumber,NumOfTests),StuNames$(StuNumber)
```

This is an example of a common phenomenon: parallel lists of numbers and strings, with common row numbers to index related information.

Or, suppose you wanted to have a three dimensional array for 100 employees in a company. The first column is to be for names, the second for salaries, and the third for social security numbers. This common enough situation can't be programmed except with a kludge (use the STR$ command). You probably would prefer to set up three parallel lists:

```
DIM Names$(100),SALARIES!(100),SocSec$(100)
```

one for the names, and the second for salaries, and the third for social security numbers (I use strings, to account for the dashes). Having done this you now would use the same pointer, i.e., the row number, to extract information from the three lists.

The way around this is to use a new structure called a Record. Records were not part of traditional BASICs, although they are common in programming languages such as C or Pascal. Essentially, a record is a type of mixed variable that you create as needed. It usually mixes different kinds of numbers and strings.

Before I can show you what a record is, I need to introduce a new type of string variable: strings of fixed length. These are set up with a variant on the DIM command. For example:

```
DIM ShortString AS STRING*10
```

This sets up a string variable, in spite of not using the $ identifier. However, this variable can only hold strings of length 10 or less. If you assign a longer string to ShortString:

```
ShortString = "antidisestablishment"
```

then what you get is:

```
ShortString = "antidisest"
```

i.e., it's truncated on the right. Similarly, if you assign a shorter string to ShortString:

```
ShortString = "a"
```

then you still get a string of length 10:

```
"a         "
```

i.e., fixed length strings are right-padded, if necessary.

Now to records. While QuickBASIC 4.5 will let you avoid maintaining parallel structures, like the employee list given above, there are costs. The major cost is that you must decide how long a name can be in the worst case. And, if you turn out to be wrong, the extra letters will be lost. This is because you can only use fixed length strings in records. A minor cost is that you may end up wasting a lot of space in memory if, say, one name had length 39 and all the others were less

than 20. QB would have to set aside space as if all the names were 39 characters long.
However, here's the first step; enter:

```
TYPE VitalInfo
    Name AS STRING * 50
    Salary AS LONG
    SocSecNumber AS STRING * 11
END TYPE
```

This defines the type. From this point on in the program it's just as good a variable type for variables as single precision or double precision.

Now to make a single variable of "type" VitalInfo, say:

```
DIM YourName AS VitalInfo
```

This variant on the DIM command sets up a single mixed variable, usually, you would say, YourName is a record variable of TYPE VitalInfo. Now you use a dot (period) to isolate the parts of this record:

```
YourName.Name = "Howard"
YourName.Salary = 100000
YourName.SocSecNumber = "036-78-9987"
```

You can also set up an array of these record variables:

```
DIM CompanyRecord(1 TO 75) AS VitalInfo
```

This sets up a list capable of holding 75 of these records. Now you can fill this array with a loop:

```
FOR I = 1 TO 75
    PRINT "Name of";I;"'th employee"
    INPUT Company Record(I).Name
    INPUT "Salary";CompanyRecord(I).Salary
    INPUT "Social security number"; CompanyName.SocSecNumber
NEXT I
```

Note the periods for each component, or element, of the record. One can even have a component of a record be a record itself. For example, we could make up a RecordOfSalary type, to keep track of monthly earnings along with the previous year's salary.

```
TYPE RecordOfSalary
    SalInJan AS INTEGER
    SalInFeb AS INTEGER
    SalInMar AS INTEGER
    SalInApr AS INTEGER
```

```
        SalInMay AS INTEGER
        SalInJun AS INTEGER
        SalInJul AS INTEGER
        SalInAug AS INTEGER
        SalInSep AS INTEGER
        SalInOct AS INTEGER
        SalInNov AS INTEGER
        SalInDec AS INTEGER
        SalInPrevYear AS LONG
    END TYPE
```

Now we can set up a "record of records":

```
    TYPE ExpandVitalInfo
        Name AS STRING * 50
        Salary AS RecordOfSalary
        SocSecNumber AS STRING * 11
    END TYPE
```

Of course, filling out all the information needed for a single record is now that much harder. Filling in the record RecordOfSalary for a single employee requires at least 13 lines of code, so filling in a record of type ExpandVitalInfo requires at least 15. It also gets a little messy to refer to the information in ExpandedVitalInfo. You thread your way down by using more periods. After a

```
    DIM GaryStats AS ExpandedVitalInfo
```

to set up a variable of this new type, use a statement like:

```
    PRINT GaryStats.Salary.SalInPrevYrSal
```

to display the information on your previous year's salary.

I should point out that your life is made a little bit easier because you can SWAP one record with another, or assign one record to another, if they are both of the same type. This means that they have been DIMmed in exactly the same way. You cannot swap or assign a variable of type VitalInfo with or to one of type ExpandedVitalInfo.

You can have records as one of the parameters in FUNCTIONs or subprograms:

```
    SUB AnalyzeSalary(X AS ExpandedVitalInfo)
```

would be the first line of a procedure that allows (and requires, in fact) parameters of TYPE ExpandedVitalInfo to be passed to it. Now you can CALL it:

```
    CALL AnalyzeSalary(GaryStats)
```

to massage your salary information. You can also pass individual components of a record whenever they match the type of the parameter.

You're probably wondering why the fuss about records. After all, parallel lists are not that difficult to manage. And, they can even be an advantage: you're not restricted to fixed length strings when using them. The short answer to this is that records are amazingly useful when dealing with random access files. This is the subject of chapter 9.

The other reason is that new versions of QB will undoubtedly go further with records. Languages such as C and Pascal have shown that records can be the most important data structure, computerese for way of organizing information, in a program. QB 5.0 (5.5 or . . .) may, I hope, eliminate many of the hassles and problems involved in programming with records. In this hypothetical version of QuickBASIC, records would be vital to good programming.

Finally, let me mention that you can use the DIM . . . AS to set up other variables. For example,

```
DIM A(1 TO 10) AS INTEGER:DIM Age AS INTEGER
```

sets up an integer array A with a range of 1 to 10 and an integer variable A. This is less confining than: DEFINT A: DIM A(1 TO 10). One peculiarity of using this form to specify the type of variables, is that you must now also use the AS clause whenever you share the variable in a procedure or function. You must say: SHARED A( ) AS INTEGER, rather than SHARED A( )—don't worry if you forget, the SmartEditor will remind you.

7 CHAPTER

Recursion

Recursion is a general method of solving problems by reducing them to simpler problems of a similar type. For the experienced programmer, it presents a unique perspective on certain problems, often leading to particularly elegant solutions and, therefore, equally elegant programs.

If you are an experienced BASIC programmer than you may be wondering why a technique that I claim is so important has not been shown to you before. The answer is simple: QuickBASIC 4.0, and now, of course 4.5, was one of the first BASICs in widespread use to allow recursion.

Recursion is sometimes thought of as a mysterious, even mystical, process, but this reputation is undeserved. See, for example, the Pulitzer prize winning book: Gödel, Escher, Bach by Douglas Hofstadter, a book some people swear by, others swear at—I oscillate between the extremes.

In fact, one typical example of recursive problem solving seems innate, at least with children. I have never met a three-year-old who has not intuitively known how to solve:

PROBLEM:	How do I deal with my parents?
SOLUTION:	Deal with father first
	Then deal with mother.

This method of solving a problem is, naturally enough, called divide and conquer, and clearly has a long history.

For a more serious example of divide and conquer, consider the following old problem: you have 7 balls and a balance scale. One ball is heavier than the other six. Find the heaviest ball in just two weighings. To solve this, first try to solve a simpler case: 3 balls. Notice that, if you try to balance two balls, then there are only two possibilities:

1. They balance, in which case the remaining ball is the heaviest.
2. They don't, in which case the heaviest one is obvious.

Now to do the seven balls problem divide the balls into two groups of three with one left over. If they balance, then as before the heaviest one is the one left over. If not then whichever side is heavier is also obvious. And, we've reduced it to the previous case. Similarly, you can do 15 balls in 3 weighings, 31 in four, and so forth.

As a final example of divide and conquer here's an outline of a recursive method for sorting called Merge sort that you'll see in section 4. It follows this outline:

```
TO SORT a LIST
    IF a list has one entry, stop.
Otherwise:
    SORT(the first half)
    SORT(the second half)
    Combine or merge the two.
```

As long as the operation of combining the two takes substantially less time than the sorting process we have a viable method of sorting. As you'll soon see, it does, and we do.

Not all examples of recursion are examples of divide and conquer: you can have recursive definitions:

Your descendants: Your children, or a descendent of one of your children.

It's recursive because it uses itself—it may even seem circular. However it's not circular because the circle returns to a different place—we define descendent of yourself using descendants of your children, or:

A counting number is: either 1, or a number that is one more than a counting number.

So 2 is a counting number because it's one more than 1, 3 is because it's one more than 2 and so on.

Similarly, the old proverb of:

"A journey of a thousand miles must begin with a single step," is a recursive solution to the "journey" problem:

```
TO Journey(1000 miles);
    Journey 1 mile, then
    Journey 999 miles.
```

It's worth pointing out that a recursive solution to a problem will always follow this outline:

```
Solve Recursively (Problem):
    IF the problem is trivial, do the obvious.
    Simplify the problem.
    Solve Recursively(simpler problem).
    Possibly combine the solution to the simpler problems(s) into a solution of the
        original program.
```

A recursive procedure constantly calls itself, each time in a simpler situation, until it gets to the trivial case, at which point it stops. There's also indirect recursion, where a function calls itself via an intermediary: function A calls function B, which in turn calls function C, which calls function A. So I should have written the outline for Journey as:

```
Journey(How many miles)
    Journey one mile: If done stop (and rest)
    ELSE Journey (one fewer miles)
```

Recursive functions

You have learned that a function can call another function—nested function calls. Recursion occurs when the function calls itself. Before I get to recursion, stop and think for a second what the compiler must do when one function calls another. Obviously, it has to communicate the current value of all parameter variables to the new function. What it does is copy the values, or the locations of the variables, to a reserved area in its memory called the *stack*. Now suppose this second function needs the results from a third function. This requires yet another copying of values of variables, and so on. However this copying can take place regardless of the nature of the other functions. It is this that makes recursion possible.

Here's an example: I introduced in chapter 1 the notion of the factorial of a positive integer. Recalls it's the product of the numbers from 1 up to the integer, and the custom is to use the ! to symbolize it. For example:

```
2! = 2*1         (=2)
3! = 3*2*1       (=6)
4! = 4*3*2*1     (=24)
```

and so on.

Here is a recursive version of a definition of the factorial:

```
DECLARE FUNCTION Factorial& (N&)
DEFLNG A-Z

FUNCTION Factorial (N)

 IF N <= 1 THEN
  Factorial = 1
 ELSE
  Factorial = N * Factorial(N - 1)      'note the call to itself -
 END IF                                 'in a simpler situation

END FUNCTION
```

Suppose I now say: PRINT Factorial(4). Then the compiler does the following:

1. It calls the function with n = 4. The first statement processed is the IF-THEN test. Since the IF clause is false, it processes the ELSE clause.
2. This says compute 4*Factorial(3).
3. It tries to compute Factorial(3). And so it now has to start building up its stack. The stack will hold partial results—those obtained to date. What gets "pushed" onto the stack can be thought of as a little card containing the status and values of all the variables as well as what is still left up in the air. In this case the card would say: N = 3. Need to compute 4 * (an as yet unknown number = Factorial(3)).
4. Now it repeats the process, calling the factorial function with a variable now having the value 3. And so another "card" gets pushed onto the stack: N = 2. Need to compute 3* (an as yet unknown number = Factorial(2))
5. Repeat the process again, so the stack contains three cards.

N = 1. Need to compute 2 * (an as yet unknown number Factorial(1)
N = 2. Need to compute 3 * (an as yet unknown number Factorial(2)
N = 3. Need to compute 4 * (an as yet unknown number Factorial(3)

Now it does one final call, with the variable N having the value 1, and sets up a fourth card. But at this point the process can stop—the top card no longer contains an unknown quantity. By the first clause, Factorial(1) is 1 so we can start popping the stack. The results of the top card, the number 1, feed into the second card. Now it can figure out what the second card stands for (it's the number 2) so it can "pop" the stack one more time, and feed the information accumulated to the third card (the number 6). Finally, it feeds the results to the bottom card and comes out with 24, or 4 factorial. Because the stack is empty, this is the answer.

The explanation of the process takes much longer than the actual solution via the QuickBASIC compiler. QuickBASIC keeps track of the partial results of any operation on its stack, and you need not be aware of the stack, most of the time.

The only time you do become aware of the stack is when it overflows and your program crashes, or behaves erratically. To prevent this, you can increase the room QuickBASIC sets aside for the stack. This is done with the CLEAR command in the form of:

```
CLEAR , , StackSize
```

Obviously it's not a good idea to wait until your program behaves strangely before you increase the size of the stack. Especially when you're developing a program it's a good idea to monitor the amount of stack space—have the computer notify you when the stack is getting tight. The way to do this is to use a variant on the FRE( ) command. FRE(−2) tells you the amount of room left on the stack:

```
IF FRE(-2) < 100 THEN PRINT "Help! tight stack."
```

More generally, you can insert breakpoints and single step through your program so that you can monitor the stack, via PRINT FRE(−2) commands from the Immediate window.

Unfortunately, this information can't be used immediately. The statement:

```
IF FRE(-2) < 100 THEN CLEAR , , 5000
```

has the unfortunate side effect of resetting all your variables back to zero, or the null string. The moral is that you should make the stack increaser the first executable statement in your program. Another moral is: monitor the stack while you're developing the program, and make sure there's enough room before making it stand alone.

There are many other examples of recursive functions. For example the Fibonacci numbers are defined as follows:

1. The first Fibonacci number is 1 (in symbols: Fib(1) = 1).
2. The second one is also 1 (in symbols FIB(2) = 1).
3. From that point on the next Fibonacci number is the sum of the two preceding ones (in symbols FIB(n) = FIB(n−1) + FIB(n−2)).

For example,

FIB(3) = FIB(2) + FIB(1) (= 1 + 1 = 2)
FIB(4) = FIB(3) + FIB(2) (= 2 + 1 = 3)
FIB(5) = FIB(4) + FIB(3) (= 3 + 2 = 5),

and so on.

The recursive definition of the Fibonacci numbers is almost trivial:

```
DECLARE FUNCTION Fib% (N%)
DEFINT A-Z

FUNCTION Fib (N)
```

```
 IF N <= 2 THEN
   Fib = 1
 ELSE
   Fib = Fib(N - 1) + Fib(N - 2)
 END IF

END FUNCTION
```

Note the pattern: the simple case is taken care of first. This is followed by reducing the calculation to a simpler case. Finally, combine the results of the simpler case(s) to finish the definition.

I should point out that, however elegant this may seem, it turns out to be an incredibly inefficient way to calculate these numbers. See the last section for more on this. I also arbitrarily defined the Fibonacci numbers at negative n to be one.

Some people called Elliot wave theorists claim the stock market follows patterns generated by Fibonacci numbers—a claim on a par with fear of the number 13. In any case, Fibonacci numbers arose when an early mathematician, Leonardo of Pisa[1] tried to model the following problem:

> "A man has one pair of rabbits at a certain place entirely surrounded by a wall. We wish to know how many pairs can be bred from it in one year, if the nature of these rabbits is that they breed every month one other pair and begin to breed in the second month after their birth."[2]

As a final example of a recursive function, consider the calculation of the greatest common divisor (GCD) of two numbers. For those who have forgotten their high school mathematics, this is defined as the largest number that divides both of them. So:
gcd(4,6) = 2: because 2 is the largest number that divides both 4 and 6.
gcd(12,7) = 1: because 1 is the largest "common divisor."

Around 2000 years ago Euclid gave the following method of computing the gcd of two numbers, a and b:

If b divides a then the gcd is b. Otherwise:

gcd(a,b) = gcd(b, a MOD b)[3]

Recall the MOD function gives the remainder you get by dividing b into a—it's obviously less than b. If a MOD b is 0, then b divides a. Here is this recursive

[1](1179? - 1240?). His pen name was Fibonacci and he was the first to introduce Arabic numbers to Europe, e.g. 0, and positional notation.

[2]"A source book in mathematics," D.J. Struik (ed.). Harvard Univ. Press 1969 p2-3.

[3]This is usually called the Euclidean Algorithm. As I mentioned in the last chapter, algorithms are what programming is ultimately about. More precisely, an algorithm is a method of solving a problem that is both precise (no ambiguity allowed) and finite—the method must not go on forever. In the case of the Euclidean algorithm, since the MOD operation shrinks the number each time, the process must stop.

outline translated into a QuickBASIC function:

```
DECLARE FUNCTION GCD% (P%, Q%)
DEFINT A-Z

FUNCTION GCD (P, Q)

  IF Q MOD P = 0 THEN
    GCD = P
  ELSE
    GCD = GCD(Q, P MOD Q)
  END IF

END FUNCTION
```

Here the pattern is a trivial case, followed by a reduction to a simpler case—no need to combine results. Since the MOD function is not restricted to short integers, it's easy enough to change the function to work with long integers as well.

Simple recursive procedures

Just as you can have recursive functions, you can have, through the magic of the stack, recursive procedures. A good example of this is a rewritten version of the binary search method for looking through an ordered list:

```
If list has length one
    then check directly (the simple case)
Else
    look at the middle of the list:
IF middle entry is too big then
    search the first half
ELSE
    search the second half.
```

Note that this outline for a recursive solution is quite close to the intuitive notion of how to search. Here is the outline translated to a procedure:

```
DECLARE SUB RecursiveBinSearch (X$, A$(), low%, high%)

SUB RecursiveBinSearch (X$, A$(), low, high)

'LOCAL variable is: Mid

   IF low > high THEN                          'A
     PRINT "Target not found"
     EXIT SUB
   END IF

  mid = (low + high) \ 2
  IF A$(mid) = X$ THEN
    PRINT "Target found at the"; mid; "entry"          'B
  ELSEIF A$(mid) > X$ THEN
   t = mid - 1
   CALL RecursiveBinSearch(X$, A$(), low, mid - 1)     'C
  ELSE
   CALL RecursiveBinSearch(X$, A$(), mid + 1, high)
 END IF

END SUB
```

A) If the list is empty, there's nothing to do.
B) Ditto if we've found the entry.
C) Again the essence of recursion—doing the same thing in a simple case.

How can you test this? Well, the first thing to do is write the appropriate module level code to set up an ordered list:

```
DEFINT A-Z
DIM B$ (255)
    FOR i = 1 TO 255
        B$(i) = CHR$(i)
    NEXT
    B$ (105) = "it"
```

Now try this:

```
CALL RecursiveBinSearch("it", B$( ), 1, 255)
```

and

```
CALL RecursiveBinSearch("itt", B$( ), 1, 255)
```

Note that I didn't bother enlarging the stack. Binary recursive search can't possibly use up much of the stack.

Whenever you're trying to understand a recursive program it's a good idea to think about what is on the stack, and what happens when the stack is finally popped. In this case each "card" on the stack contains:

1. The address of the array A$( ) (arrays are always passed by reference).
2. The current values of the variables low and high, and a new copy of the local variable Mid.

These are also the keys when debugging a recursive procedure. After all, WATCHing variables on the stack is useless if you don't know what they're supposed to do.

By now you may be thinking that recursion is just a fancy way of avoiding loops. And, there's some truth to this (see the last section), but there are many problems for which it would be hard to find the loop equivalent. The simplest example of this is the famous Tower of Hanoi problem. As the quote that follows indicates, it was originally called the "Tower of Brahma." Here's the problem:

> "In the great temple at Benares rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles God places sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disc at a time and that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish"[4]

I'll continue to use 64 discs in the explanation that follows, but solving this problem with this many discs would take the priests, or a super computer for that matter, more time than scientists say the universe has been around, or is likely to be around. For n disks the solution takes 2^n − 1 steps, for 64 disks this has 19 digits, approximately 1.844674E+19, according to QB.

Anyway, to solve this problem recursively we need to decide on the trivial case: it's when the tower is down to one disc, height one. Next, we have to find a way to simplify the problem while retaining the same form, and making sure that this process does eventually lead to the trivial case. The key is to note that the bottom disc is irrelevant when we move the first 63 discs from the first tower to any other tower. Since it's larger than any disc we can just as well regard a peg with it alone as being empty when it comes to moving discs around. Next, note that we can change the destination temporarily, if it helps simplify the problem. Given all this, here's an outline of a solution to the "Tower of Hanoi":

Move the top 63 discs to tower 2 using tower 3.
Move the bottom disc to its destination (tower 3).
Move the top 62 discs to tower 1 using tower 3. Of course, to move 62 disks is a problem of smaller size. And, just as before, the bottom disk will be irrelevant.

Here's a procedure with some module level code that implements this outline:

```
DECLARE SUB SolveTowerOfHanoi(Hght!, FromTowr!, ToTowr!,UsingTowr!)
CLEAR , , 10000                    ' Needs a lot of stack room!

CLS
INPUT "How many discs - don't try too many!"; NumberOfDiscs
CALL SolveTowerOfHanoi(NumberOfDiscs, 1, 2, 3)
END

SUB SolveTowerOfHanoi (Height, FromTower, ToTower, UsingTower)
```

[4]"Mathematical recreations and essays," W.W. Rouse Ball and H.S.M. Coxeter. Dover 1987 p317.

```
  IF Height = 1 THEN
   PRINT "Move a disc from tower #"; FromTower; "to tower#";
ToTower
  ELSE
   CALL SolveTowerOfHanoi(Height - 1, FromTower, UsingTower,
ToTower)
   PRINT "Move a disc from tower #"; FromTower; "to tower #";
ToTower
   CALL SolveTowerOfHanoi(Height - 1, UsingTower, ToTower,
FromTower)
 END IF

END SUB
```

As mentioned in the outline the key to this solution is the switch in the destinations between:

```
     CALL SolveTowerOfHanoi(Height – 1,FromTower,UsingTower,ToTower)
```

and

```
     CALL SolveTowerOfHanoi(Height – 1,UsingTower,ToTower,FromTower)
```

(Note: these should be entered as on a single line.) This may be confusing; if so, I suggest analyzing the stack for the simple case of three discs by playing computer. The watch window can act as a crib if necessary.

Two more complicated recursions

Suppose you're stuck in a maze. At any point you can go in one of four directions: north, east, south, and west. Some of these directions may be blocked at some places in the maze. How do you find your way out? Obviously, you'd try to analyze your way out. A computer program doesn't have this option, but it does work quickly. Quickly enough so that sophisticated trial and error can lead to a way out. In fact, given the speed of QuickBASIC, there's an easy recursive solution to getting out of any maze:

1. Mark the square you're on
2. If you can go north do so, and mark that square
3. If not go east, and mark that square, etc.

By marking the squares you've made the problem simpler—there's one less square to worry about. Because eventually you either find a way out, or get stuck, this outline fulfills the pattern for a recursive solution.

In the following procedure I've assumed that the maze is stored as an array of characters with the barrier being marked with a string variable called Barrier$, the empty squares marked with a character I'll call NotUsed$, and the goal marked with a "!". For a large maze you might need to store it as an array of integers, to save space.

First off, this is the procedure:

```
SUB FindExit (Row, Col)
SHARED Maze$(), Finished, Barrier$, NotUsed$
STATIC CharNum                                  'A

CALL PrintMaze
CharNum = CharNum + 1                      'B
IF CharNum = 89 THEN CharNum = 0

  MarkChar$ = CHR$(33 +CharNum)
  Maze$(Row, Col) = MarkChar$

 ' these blocks do the recursion 4 blocks = 4 ways to travel

     IF NOT Finished THEN                       'C
       SELECT CASE Maze$(Row, Col - 1)
         CASE NotUsed$
          CALL FindExit(Row, Col - 1)           'try to go west
         CASE "!"
          Finished = True
       END SELECT
     END IF

    IF NOT Finished THEN
      SELECT CASE Maze$(Row - 1, Col)
        CASE NotUsed$
         CALL FindExit(Row - 1, Col)
        CASE "!"
         Finished = True
      END SELECT
    END IF                                                 'North

    IF NOT Finished THEN
      SELECT CASE Maze$(Row, Col + 1)
        CASE NotUsed$
          CALL FindExit(Row, Col + 1)
        CASE "!"
         Finished = True
      END SELECT
    END IF

    IF NOT Finished THEN
      SELECT CASE Maze$(Row + 1, Col)
        CASE NotUsed$
         CALL FindExit(Row + 1, Col)
        CASE "!"
         Finished = True
     END SELECT
    END IF

 END SUB
```

A) I mark each square with a different ASCII character. By making CharNum a STATIC variable I ensure that on each recursive pass a different marker is used. Moreover, since at each call I'm going to print the current status of the maze, with a call to that SUB, using successive ASCII characters as markers makes my progress from step to step clearer.
B) I don't want to go to the higher ASCII characters, especially because I'm using CHR$(219) (=■) as the boundary marker.
C) This and its three sibling blocks is the key to the recursion. I use a SHARED flag variable, Finished, to mark when I've succeeded. Since it's SHARED, any changes I make remain as QB pops the stack. (A local variable would not work. Would a STATIC variable work?) If you are still confused as to what this block is doing, imagine that you are a mouse sitting at a square. You look to your left. If that square hasn't been marked you move to it, checking if it's your goal. If it's not, you start over. If going to your left doesn't work then you try going up, and so forth.

The main module for this procedure might look like this:

```
DECLARE SUB Initialize (Size%)
DECLARE SUB FindExit (Row%, Col%)
DECLARE SUB PrintMaze ()
CLEAR , , 32000               ' needs maximal stack size

DEFINT A-Z                   ' integers speed things up

CONST True = -1
CONST False = 0
NotUsed$ = ""
Barrier$ = CHR$(219)          ' = █
Finished = False
CLS

INPUT "How big is the maze";Size
DIM SHARED Maze$(Size,Size)              'A
CALL Initialize(Size)                        'B

CLS
PRINT "Enter starting row and column"
INPUT "row"; StartRow
INPUT "column"; StartCol
CALL PrintMaze                          'C

CALL FindExit(StartRow, StartCol)  ' where to start

IF Finished THEN                        'D
  CALL PrintMaze
ELSE               'gets here if all squares used up
 BEEP
 CLS
 PRINT "No way Out!"
END IF
```

A) This is another variant on the DIM command. It indicates that the Maze$ array will be a SHARED or global variable for all the procedures in this program.
B) This sub-program will set up the maze.
C) Printing the maze is an obvious candidate for a SUB.
D) Again, because Finished is a SHARED variable, I know its value reflects whether the FindExit worked or not.

Now to the Initialize SUB. The most elegant way to set up the maze is to use a variant on the cursor mover program from chapter 8, but for illustration purposes I've set it up using a very simple 10×10 maze created with a loop.

```
SUB Initialize(Size)

SHARED Maze$(), Barrier$

Size = 10

FOR I = 0 TO Size
  Maze$(0, I) = Barrier$
  Maze$(Size, I) = Barrier$
  Maze$(5, I) = Barrier$
  Maze$(I, 0) = Barrier$
  Maze$(I, Size) = Barrier$
NEXT I

Maze$(9, 5) = "!"
Maze$(5, 2) = NotUsed$                   'need to leave a way out!

END SUB


' SUB to print the Maze

SUB PrintMaze

CLS

 SHARED Maze$(),Size

 FOR I = 0 TO Size
  FOR J = 0 TO Size
   PRINT USING "\\"; Maze$(I, J);
  NEXT J
  PRINT
 NEXT I

 END SUB
```

One way to understand this procedure is to imagine a tree, sometimes called a decision tree. From any given square you have multiple branches. The branches continue blossoming out until the procedure gets stuck. Then it backtracks to where it started and goes north. More precisely, starting from any square the procedure systematically exhausts all the possible squares it can get to by first going east—leaving a trail of used up squares. If one of these possibilities leads to an exit, the procedure ends. The flag given by the shared variable Finished flips to true. Otherwise, the procedure backtracks all the way back to the starting square and tries to go north. It can do this only if the square directly east of the starting square is not already marked. Similarly, if it hasn't found an exit going north, or it couldn't get started because the east square was marked, it tries east and south. If none of these directions are possible, then the maze had no exit and the procedure ends.

As you can well imagine this process can take time. You might want to experiment with larger mazes, but be prepared to change Maze$ to an array of integers. Note that each square searched can add a "card" to the stack. (If you run it with a large maze why not monitor the changing free stack space?) In general the "cards" on the stack contain the current value of the row and column, and the address of the array. Each call to the procedure makes a permanent change to the array. Without making the array a shared variable there would be no way to keep track of where you've been.

The final example I want to give in this section is much shorter, but it's very tricky. Before I do, though, recall that each time you run the jumble program you'll probably get a different arrangement of the original string. Suppose, however, that you wanted to generate all possible jumbles (arrangements). (The technical term for a jumble is a permutation. For a string of length "n" there are n! or n factorial different ones.) The key to programming this is to note that to obtain all the permutations, say, of the string "abcd" you need only follow the outline:

List "a" concatenated (joined with the plus sign) to all permutations of "bcd."
List "b" concatenated with all permutations of "acd."
List "c" concatenated with all permutations of "abd," and finally:
List "d" concatenated with all permutations of "abc."

Now to get the permutations of any of the three letter combinations you repeat the process:
"b" together with all the permutations of "cd"
"c" with the permutations of "bd"
"d" with the permutation of "bc."

You stop when you are working with a string of length one.

The problem with following this outline is that it's easier to do on paper than by computer. Somehow you have to keep track of which characters you've switched. This means each time you call the procedure, you need a different version of the string variable you're permuting—you want the procedure to play with

a different value on each "card" in the stack. On the other hand, you need to preserve the identity of the original string, if only to move the letters in it around. All this means that you want to pass by value rather than by reference. Next, you need to keep track of which letter you've switched, so as to know when to stop. The best way to do this is to make the procedure depend on two parameters with the second, an integer, giving the current position within the string.
This is the procedure:

```
SUB Permute(A$,n)

'LOCAL variables LenA,Temp$,First$,I

LenA = LEN(A$)

IF n = LenA THEN
  PRINT A$                          'A
ELSE
  FOR I = n TO LenA
      CALL Switch(A$,n,I)           'B
      CALL Permute((A$),n+1)        'C
    NEXT I
END IF

END SUB
```

A) When n is the length of A, there's nothing left to do.
B) This procedure interchanges two characters—it's pretty easy:

```
SUB Switch(A$,I,J)
     ' LOCAL variable T$
          T$ = MID$(A$,I,1)
          MID$(A$,I,1) = MID$(A$,J,1)
          MID$(A$,J,1) = T$
END SUB
```

   Note however that since you want the changes to persist you pass by reference here. I made it a separate procedure because I prefer to keep procedures to single tasks.
C) On the other hand, here you want the stack to contain a new copy of the variable each time it is called, so you pass by value. You only want to switch the letters in the new value of the variable. Here again it's a good idea to draw a tree that shows what follows what, and what the values of the various variables are.

   To use this procedure you only have to enter:

```
CALL Permute(X$,1)
```

Recursive sorts

So far you've seen three useful methods for sorting: insertion, ripple, and Shell sort. Insertion and ripple sort are good for short lists, and Shell sort is good for moderately sized lists. The sorts you'll see in this section are among the fastest known: they are the ones of choice for very large lists.

The first one, merge sort, has the easiest outline—you saw it in the introduction:

```
TO SORT a LIST
    IF a list has one entry, stop.
    Otherwise:

        SORT the first half
        SORT the second half
        Combine and merge the two.
```

The procedure to sort a list, once you write the merge procedure, is almost trivial:

```
SUB MergeSort (a$(), Start, Finish)

  'LOCAL  Variable is Mid

  IF Start < Finish THEN
    Mid = (Start + Finish) \ 2
    CALL MergeSort(a$(), Start, Mid)
    CALL MergeSort(a$(), Mid + 1, Finish)
    CALL Merge(a$(), Start, Mid, Finish)
  END IF

END SUB
```

This procedure keeps on splitting the list: when it gets to n lists of size one, the Merge procedure will combine them into n/2 ordered lists of size two, n/4 ordered lists of size four, n/8 ordered lists of size 8, and so on. At this point I've really swept the details under the rug by moving them to the as yet unwritten Merge procedure.

Merging two ordered files, or ordered parts of the same file, is intuitively obvious, but a bit tricky to program. What you have to do is set up a temporary array and work your way slowly through the lists, filling up the temporary array with the appropriate entry from one of the two lists. When you're done you have to write the temporary array back to the original array. Here's the Merge procedure:

```
SUB Merge (a$(), Start, Mid, Finish)

'LOCAL variables are:
'Temp$(),Begin1,End1,Begin2,End2,TempLocation,I
DIM Temp$(Start TO Finish)
```

```
Begin1 = Start
End1 = Mid
Begin2 = End1 + 1
End2 = Finish

TempLocation = Start

DO WHILE Begin1 <= End1 AND Begin2 <= End2              'A
  IF a$(Begin1) <= a$(Begin2) THEN
    Temp$(TempLocation) = a$(Begin1)
    TempLocation = TempLocation + 1
    Begin1 = Begin1 + 1
  ELSE
    Temp$(TempLocation) = a$(Begin2)
    TempLocation = TempLocation + 1
    Begin2 = Begin2 + 1
  END IF
LOOP

IF Begin1 <= End1 THEN                                  'B
   FOR I = Begin1 TO End1
    Temp$(TempLocation) = a$(I)
    TempLocation = TempLocation + 1
   NEXT I
ELSEIF Begin2 <= End2 THEN
   FOR I = Begin2 TO End2
    Temp$(TempLocation) = a$(I)
    TempLocation = TempLocation + 1
   NEXT I
END IF

FOR I = Start TO Finish                                 'C
 a$(I) = Temp$(I)
NEXT I

END SUB
```

A) This loop runs through the list. It systematically compares entries in the two, and moves the smaller one to the temporary list. After every move, it shifts a pointer (TempLocation) that moves within the temporary list one step forward. Similarly, it moves a pointer within a given sub-list, either Begin1 or Begin2, whenever it does a swap. The loop constantly checks the status of these pointers to avoid going past the boundaries of the individual sub-lists.

B) You get to here when one of the sub-lists is used up. This block copies the remainder of the other list to the temporary array.

C) This copies the temporary array back to the original array. Without doing this, the recursion would fail.

Although merge sort is theoretically one of the fastest sorts, in practice the naive formulation given above is too slow. Sorting a list of 1000 random four letter strings takes about 6 1/4 minutes, 436.71 seconds on a basic PC, slower even than insertion sort which, you may recall takes about 4 minutes. And far slower than Shell sort, which takes about 17 seconds for this problem. However, unlike insertion sort, doubling the size of the list no longer quadruples the time—it slightly more than doubles it. So even this naive formulation of merge sort will be much faster than insertion sort, for a list of 2000 items. Shell sort still remains much faster: it takes 41 seconds.

One problem is that copying the temporary array back to the original list takes too much time. Unfortunately, there's little you can do about this. (Wouldn't it be nice to have a built-in SWAP for lists?)

On the other hand the procedure also spends too much time and stack space on the trivial cases of lists of size one and two. You can dramatically speed up Merge sort and save a lot of stack space by modifying what the procedure regards as the trivial case. For example, suppose you directly sort all lists of length one or two by swapping entries as needed. Change the original procedure to read like this:

```
SUB MergeSort(A$(1),Start,Finish)

  'LOCAL variable is Mid

  IF  Finish - Start <= 1 THEN
   IF A$(Finish) < A$(Start) THEN SWAP A$(Finish),A$(Start)
  ELSE
    MID = (Start + Finish)\2
    CALL MergeSort(A$(),Start,Mid)
    CALL MergeSort(A$(),Mid+1,Finish)
    CALL Merge(A$(),Start,Mid,Finish)
  END IF

END SUB
```

Now you are directly swapping the entries when the lists are tiny. The savings are dramatic: sorting 1000 random four letter combinations now only takes about 3 3/4 minutes on a basic PC, about half as much time, and now it's slightly faster than insertion sort. Even more dramatic savings result by modifying the trivial case even further. Recall that insertion or ripple sort is very fast for small lists, say lists of size 64 or less. If you add an insertion sort procedure to a program containing a merge sort, and rewrite the fundamental procedure as here:

```
  IF  Finish - Start <= 63 THEN
   CALL InsertSort(A$(),Start,Finish)
  ELSE
    MID = (Start + Finish)\2
    CALL MergeSort(A$(),Start,Mid)
    CALL MergeSort(A$(),Mid+1,Finish)
    CALL Merge(A$(),Start,Mid,Finish)
  END IF

END SUB
```

and the improvements are incredible. Sorting a list of 1000 four letter strings now takes less than 1/2 minute on a basic PC, and sorting a list of 2000 strings takes less than 1 minute (56.61 seconds). I decided to use 64 items as the trivial case by experimenting—it gave the best results.

This modified Merge is now almost as fast as Shell sort and is now faster than Shell sort once the list has more than, say, 3000 entries, and is roughly the same for lists of size 2200-3000. Thus combining insertion sort with merge sort gives you a fast sort for very large lists. In theory you could modify merge sort to be even faster by using Shell sort for lists of moderate size, but I won't bother presenting that version here. In any case all these tweaks preserve the essential advantage of the original merge sort—doubling the list still only slightly more than doubles the time needed. Unfortunately, all the versions of merge sort do have one big disadvantage, and ironically this disadvantage shows up only for the very large lists on which merge sort should shine: you need twice as much space as is needed for Shell sort, because of the temporary array used in the merge procedure. This means that you're likely to run out of space for very large lists. So Merge sort is of more theoretical than practical interest in most situations.

As many people have remarked, finding a better general purpose sort is the better mousetrap of computer science. Unfortunately, the best general purpose sort currently known, usually called Quick Sort,[5] unlike the various modifications of merge sort, is not guaranteed to work fast. In very unlikely situations it can be the slowest sort of all.

If merge sort is a "divide and conquer" recursion, then quick sort can be thought of as a "conquer by dividing" recursion. To see what I mean, consider the list of numbers:

    5,12,4,9,17,21,19,41,39

The number 17 is in an enviable place: all the numbers to the left of it are smaller than it and all the numbers to the right of it are greater than it. This means that 17 is in the correct position when this list will be sorted. It partitions the list and will not have to be moved by any sort. The idea of Quicksort is to create these "splitters" artificially—on smaller and smaller lists. Here's the basic outline:

1. Take the middle entry of a list.
2. By swapping elements within the list make this element into a splitter. Note that this element may need to move, and this is obviously the most difficult part to program.
3. Divide the list into 2 at the splitter and repeat steps 1 and 2.
4. Continue until both the lists created by making a splitter have size, at most, one.

[5]Invented by the computer scientist C. A. R. Hoare around 1960.

The next program translates this outline into a procedure. All numeric variables are assumed to be integers, say by a DEFINT A-Z statement:

```
SUB QuickSort (A$(), start, finish)

'Local variable PosOfSplitter

IF finish > start THEN
 CALL Partition(A$(), start, finish, PosOfSplitter)
 CALL QuickSort(A$(), start, PosOfSplitter - 1)
 CALL QuickSort(A$(), PosOfSplitter + 1, finish)
END IF

END SUB
```

Now you need to write the procedure that forces the splitter. This procedure is subtle, and makes Quicksort harder to program than merge sort. Luckily there are many ways to do this, the one I'll show you here is inspired by insertion sort.[6] You move the splitter out of the way first. Next you start from the left end of the list and look for any entries that are smaller than the splitter. Whenever you find one you move it, keeping track of how many elements you've moved. When you get to the end of the list, this marker will tell you where to put back the splitter. Here is this procedure:

```
SUB Partition (A$(), start, finish, LocOfSplitter)

' LOCAL variables are:SplitPos,NewStart,I,Splitter$

SplitPos = (start + finish) \ 2
Splitter$ = A$(SplitPos)

SWAP A$(SplitPos), A$(start)          'get it out of the way

LeftPos = start

   FOR i = start + 1 TO finish
     IF A$(i) < Splitter$ THEN
       LeftPos = LeftPos + 1
       SWAP A$(LeftPos), A$(i)
     END IF
   NEXT i

SWAP A$(start), A$(LeftPos)           ' LeftPos marks the hole

LocOfSplitter = LeftPos                ' This gets passed to

                                      ' original procedure.
END SUB
```

[6]I think it's due to N. Lomuto. Another version of Quicksort is one of the sample programs.

Quicksort is usually quite fast. To sort a list of 2000 random four letter strings takes around 17.3 seconds. However, how fast Quicksort works completely depends on how much the splitter splits—the ideal is when it splits the list in two. If each time you sort the smaller list, the element you are trying to make into a splitter is the smallest, or the largest, in the list, then in one of the recursive calls, too little work is done; in the other, too much is done. This makes Quicksort slow down—for all practical purposes it becomes a complicated version of insertion sort. If this horrible situation should come to pass then you may end up waiting a long time to sort a list of 2000 entries. Luckily, this worst case is quite unlikely, but it can happen. To prevent this, computer scientists suggest that you:

1. Use an insertion or ripple sort for small lists, much as I did in the tweaked version of merge sort. My tests show that this works best when you use insertion sort or ripple sort on lists of size 8 or less. This speeds up the program considerably: to sort a list of 2000 random four letter strings now takes around 15 seconds—around a 15% improvement! It also saves stack space.
2. Most important: to eliminate the chance of the worst case happening, don't use the middle element as the potential splitter. One idea is to use the random number generator to find a random element on the list. Change the lines:

   ```
   SplitPos = (Start + Finish) \ 2
   Splitter$ = A$ (SplitPos)
   ```

   to be:

   ```
   SplitPos = Start + INT( (Finish -Start+1)*RND(1) )
   ```

Doing this makes it almost inconceivable that you'll end up in the worst case. The problem is that using the random number generator takes time. While it makes the worst case almost inconceivable, it does, according to my tests, slow the average case down around 25% (23.8 seconds vs. 17.3 seconds). On the other hand, if you use this idea in the tweaked version described above then the deterioration is much less. My tests show that this combination takes around 16.5 seconds. This is only a 10% reduction, and it's still faster than the original version of quicksort. Far fewer calls to the random number generator are necessary, because we're not dealing with small lists. This is my personal favorite version of Quicksort, and is the sort I use in consulting. Again, all times were done by a 4.77 MHz PC.

Another possibility that many people prefer is to keep on using Insertion or ripple sort for small lists, but instead of calling the random number generator to find a candidate for the splitter, use the median of the start of the list, the middle term and the end of the list. The code for finding the median of three items is almost trivial, and doesn't take very much time either.

Finally, why so many sorts? One reason is that they illustrate the techniques so well, but there's a more serious reason as well. Although Quicksort and Shell sort are fast, they do have one disadvantage that insertion, ripple and merge sort do not have. They are not stable. To understand what stability means, suppose you have a list of names and addresses that is already ordered alphabetically by name. Suppose now that you want to re-sort the list by city and state. Obviously you want the list to be ordered alphabetically by name within each state. Unfortunately, if you use quick or Shell sort then the alphabetical order of the names will disappear. With merge sort, you can preserve the old order within the new.

Binary trees

In many cases you don't really need to sort a list, rather you sort the list so that you can easily search through the list using binary search. Or, suppose you have a person entering random names, and you want to be able to prevent duplicates. If the list were ordered then this would be easy, but a new entry requires re-sorting the list. If you choose this option use insertion sort—it's obviously the best way to insert a new item. Finally, you may know that at some point in the future that you'll frequently need to search through the list, possibly ordering the list as well. Stopping and re-sorting the list each time the user enters a new name is obviously not the method of choice if the final ordering happens in the far future.

It should come as no surprise that programmers have long searched for methods of accepting information for a list so that you can still search through the list as fast as through an ordered list, i.e., as fast as binary search, and hopefully, at some future time, sort it quickly as well. And all this should take place without having to stop and constantly re-sort.

Of course, you can't expect to get something for nothing. The methods programmers invented require more sophisticated programs, and use a lot more space. Moreover, unless you use very sophisticated methods, which are not described here, this method works only if the names are entered in a random order. If someone applies these methods when entering an ordered list the results will be horrible.

To understand this method, called binary trees, first suppose you have a list of names:

Joan,Susan,Gary,Karie,Ted,Mike,David,Al,Harry,Hal

Start by putting the first name on the top of a piece of paper:

Joan

Since the name Susan comes in the alphabet after Joan, draw an arrow pointing to the right (it's sometimes called an "after" or "right" arrow).

$$\begin{tikzcd}
\text{Joan} \arrow[dr] & \\
 & \text{Susan}
\end{tikzcd}$$

Gary comes before Joan, so draw a left arrow (a "before" arrow).

$$\begin{tikzcd}
 & \text{Joan} \arrow[dl] \arrow[dr] & \\
\text{Gary} & & \text{Susan}
\end{tikzcd}$$

Karie comes after Joan, but before Susan, so the picture looks like this:

$$\begin{tikzcd}
 & \text{Joan} \arrow[dl] \arrow[drr] & & \\
\text{Gary} & & & \text{Susan} \arrow[dl] \\
 & & \text{Karie} &
\end{tikzcd}$$

Ted comes after Joan, and after Susan, so the picture now looks like this:

$$\begin{tikzcd}
 & \text{Joan} \arrow[dl] \arrow[drr] & & & \\
\text{Gary} & & & \text{Susan} \arrow[dl] \arrow[ddr] & \\
 & & \text{Karie} & & \\
 & & & & \text{Ted}
\end{tikzcd}$$

The idea is that when you insert a new name, follow the branches in the right order. Saul comes after Joan, before Sue, and after Karie. At the next step the picture looks like:

$$\begin{tikzcd}
 & \text{Joan} \arrow[dl] \arrow[drr] & & & \\
\text{Gary} & & & \text{Susan} \arrow[dl] \arrow[ddr] & \\
 & & \text{Karie} \arrow[dr] & & \\
 & & & \text{Saul} & \text{Ted}
\end{tikzcd}$$

Finally the picture looks like:

$$\begin{tikzcd}
 & & & & \text{Joan} \arrow[dl] \arrow[ddr] & & & \\
 & & & \text{Gary} \arrow[dll] & & & \text{Susan} \arrow[dl] \arrow[dd] \arrow[ddr] & \\
 & \text{David} \arrow[dl] \arrow[dr] & & & & \text{Karie} & & \\
\text{Al} \arrow[dr] & & \text{Harry} & & & & \text{Saul} & \text{Ted} \\
 & \text{Hal} & & & & & &
\end{tikzcd}$$

Naturally enough, programmers call the resulting diagram a binary tree. Binary because there's never more than two branches, and tree because the picture resembles an upside down tree.

It's a good idea to learn some of the terminology computer scientists use to describe binary trees. For example, they say that each of the names form, or sit in, a *node* on the binary tree. The top node is called the *root* of the tree. The node directly above that points to a node is called the *parent* of that node. So Karie is the parent node for Saul, and David is the parent node for Al and Harry. Every parent in a tree can have either no children, a younger (left) child, an older (right) child, or both. A node that has no children is said to be a *leaf*.

When you enter names, and form a binary tree by following the above procedure, then you are said to have *loaded the tree*. Adding a new name to a loaded tree is fairly quick, almost as fast as just sticking the name on at the end of an ordinary list. Although now, instead of merely sticking it on at the end, you follow the correct branch of the tree, until you can go no further. At that point stick the new name where the hole is. Note that searching a tree and loading a tree are basically the same operation.

Note as well that a binary tree is inherently a recursive object. After all, start from any node at all, then all the nodes that are attached to it form a binary tree as well with the first node as the new root, usually called a *sub-tree*. The left child of a parent gives us the left sub-tree and the right child of a parent gives us the right sub-tree. It is this recursive property that will make most of the operations on a binary tree simple.

A recursive version of loading a tree will be a procedure that depends on the contents of the current position in the tree, as well as the sub-tree. An outline in something close to pseudo-code might look like:

```
Procedure LoadATree(Tree,NodeInTree,ThingToAdd)
    IF the current node is empty, add the word there
    ELSE
    IF the word precedes the contents of the node
        CALL LoadATree(Left Tree, LeftChild, ThingToAdd)
    ELSE
        CALL LoadATree(Right Tree, RightChild, ThingToAdd)
```

To write a program that implements this, you have to decide on how you're going to represent the arrows and the holes. This is usually called the *data structure*. The ideal situation would be to use a three column array. Each row on the array would use the first column for the contents of the node, the second column for the row the left child is on, and the third column for the row the right child is pointing to.

The trouble with this is that if you use a string array, you can't use numbers for the entries in the second and third columns (the rows of the children). So you might turn to a list of records of type:

```
TYPE ContentsOfTree
    NodeContents AS STRING*?
    RowOfLftChil AS INTEGER
    RowOfRtChild AS INTEGER
END TYPE
```

The problem with this is indicated by the ?. How much room to allow for the NodeContents?

O.K., so you go back to thinking about using the string equivalent of the row numbers—converting them back and forth via VAL and STR$. In our example, the 10 by 3 array would look like this, using −1 if there's nothing to point to:

Row Number (= node number)	Contents of Node	Row "before" arrow points to	Row "after" arrow points to
1	Joan	3	2
2	Susan	4	5
3	Gary	7	9
4	Karie	−1	6
5	Ted	−1	−1
6	Mike	−1	−1
7	David	8	9
8	Al	−1	−1
9	Harry	10	−1
10	Hal	−1	−1

So for example the third row of the array that will represent the tree would contain three strings, one in each column:

"Gary" "7" "9"

Note that the first column has the entries in the order they were entered—it's the other two columns that give us the extra information needed to quickly search through the tree.

However, I prefer to use the idea of parallel lists that I talked about in the last chapter, among other reasons because it saves space, but ultimately it's a matter of taste.

So the first list will hold the contents of the nodes—it will be a string array, and I'll use a two column integer array for the pointers to the left and right children.

This is the procedure to load a tree:

```
SUB LoadAtree (nodes$(), arrows%(), item$, CurrentPos%)  'A

' LOCAL variables are:Items%,WhereToGo%,Direction$

Items% = VAL(nodes$(0))

IF Items% = 0 THEN                                      'B
  nodes$(0) = "1"
  nodes$(1) = item$
  arrows%(1, 0) = -1
  arrows%(1, 1) = -1
```

```
    EXIT SUB
  END IF

  IF item$ < nodes$(CurrentPos%) THEN
    WhereToGo% = arrows%(CurrentPos%, 0)
    Direction$ = "left"
  ELSE
    WhereToGo% = arrows%(CurrentPos%, 1)
    Direction$ = "right"
  END IF

  IF WhereToGo% = -1 THEN                                   'C
    Items% = Items% + 1
    nodes$(Items%) = item$
    arrows%(Items%, 0) = -1
    arrows%(Items%, 1) = -1
    nodes$(0) = STR$(Items%)

  ' Now change pointers on parent node

    SELECT CASE Direction$
      CASE "left"
        arrows%(CurrentPos%, 0) = Items%
      CASE "right"
        arrows%(CurrentPos%, 1) = Items%
    END SELECT
  ELSE
    CALL LoadAtree(nodes$(), arrows%(), item$, WhereToGo%)  'D
  END IF

  END SUB
```

A) As mentioned above, the contents of the nodes are stored in a string array, indicated by the parameter array Nodes$( ), with Nodes$(0) being the string equivalent of the number of items. The arrows are stored in an array Arrow%(,). The parameter Item$ is for the item to be loaded and the parameter CurentPos% is used to indicate where you are in the tree. To load the tree, CALL this procedure with CurrentPos% = 1.

B) You need a special case when you're setting up the tree. Note that I'll use the 0th position of the nodes to keep track of the number of items. This is necessary for two reasons. The first is that you have to know in what row of the array you'll add the next entry. The second is that when using this procedure, you have to have previously dimensioned the arrays—a robust program would constantly monitor this number.

C) This gives us a way out of the recursion. When the arrow is pointing to a hole, it's time to place the entry.

D) This starts the recursion.

Of course, in a more serious example, you would have to increase the dimension of the arrays to allow enough room. In any case, to print out the table you can use this program:

```
SUB PrintaTree (Tree$(), arrows%())

' LOCAL variable is NumberOfItems

CLS
PRINT "Row          Contents       Row `before'       Row
`after'"
PRINT "Number       of Node        arrow points       arrow
points"
PRINT "(= node                     to                 to"
PRINT "number)"

PRINT : PRINT

NumberOfItems = VAL(Tree$(0))

FOR I = 1 TO NumberOfItems
 PRINT I, Tree$(I), arrows%(I, 0), arrows%(I, 1)
NEXT I

END SUB
```

You might want to slow this down with some empty INPUT$(1) or SLEEP statements.

Modifying the LoadATree to SearchATree is almost trivial, and I'll leave that to you. In any case you should be aware that the efficiency of SearchATree depends on the shape of the tree. The best situation, usually called a complete or balanced tree is when the tree is as bushy as possible. In this case it works as fast as ordinary binary search. The worst case is when someone gives you an ordered list. In this case the tree is as narrow as possible and if you use SearchATree then you'll end up searching the list one item at a time.

One of the nicest features of trees is that you can sort the items in the tree quite fast—about as quickly as for quicksort. To do this you have to traverse or move through nodes cleverly. Here is an outline of how this is done; it's usually called inorder traversal:

1. Go left as far as you can. Print the node.
2. Print the parent of the node from step 1.
3. From the parent, go right, if you can. If you can go right then repeat step 1. If you can't, move up to the grandparent from step 1 and repeat step 3.

This outline obviously describes a recursive procedure, and here is one version that implements it:

```
SUB TreeSort (Nodes$(), Arrows%(), CurrentPos%)

'LOCAL variable is: WhereToGo%
```

```
 IF Arrows%(CurrentPos%, 0) <> -1 THEN
  WhereToGo% = Arrows%(CurrentPos%, 0)
  CALL TreeSort(Nodes$(), Arrows%(), WhereToGo%)
 END IF

 PRINT Nodes$(CurrentPos%)

 IF Arrows%(CurrentPos%, 1) <> -1 THEN
  WhereToGo% = Arrows%(CurrentPos%, 1)
  CALL TreeSort(Nodes$(), Arrows%(), WhereToGo%)
 END IF

END SUB
```

To use this in the example above:

```
    CALL TreeSort(Tree$( ),Arrow%( ),1)
```

The only problem with using this to sort a list is that, just like using SearchATree, its efficiency completely depends on the shape of the tree. The closer the tree is to being complete (maximum bushiness), the faster this works. On the other hand if the tree degenerates because the names entered were already ordered, then TreeSort does as well. Consult an advanced book on algorithms—see the bibliography—to learn how to make sure your trees stay balanced. This requires carefully monitoring the shape of the tree and making some rather complicated switches.

Trees do have one other problem, especially when you're using arrays to represent the contents of the nodes and the pointers: it's quite painful to delete a node. In fact, it's almost never worth the trouble to try to reclaim the space. The best method I know to delete a node is to follow the following outline—I'll leave the details to you:

1. If the node has no children, simple delete it.
2. If the node has one child, attach the parent of the node to be deleted to the lone child.
3. If the node to be deleted has two children, do the following: Go to its left child, and then as far right as possible. This is the entry that, alphabetically, comes right in front of the entry being deleted. Swap it with the entry to be deleted.

In spite of the occasional problems binary trees are an extremely useful technique. Most of the time they are the best way to build an index. As you'll see in the files chapter they are the key to writing an XREF program to cross reference your programs. Which, if you stop to think about it, is exactly the same problem as writing any other kind of index.

When not to use recursion

Many of the example programs I showed you could have been solved by iteration (writing a loop). And, to quote Niklaus Wirth, the inventor of Pascal: ". . . the lesson to be drawn is to avoid the use of recursion when there is an obvious solution by iteration."[7]

The reason is that although a recursive procedure is often shorter to write, it almost inevitably takes longer and uses much more memory to run. You may counter that memory is cheap, but no matter what you do, in QB the stack is no larger than 32K.

In fact, as he and others have pointed out, what should be the standard examples of when not to use recursion are also the examples most commonly given of recursion: the factorial and the Fibonacci numbers. O.K., I plead guilty to showing them to you, but, for example, Microsoft gives only the factorial in its manuals.

Both the factorial and recursion can be computed easier and faster, and using much less memory, using a loop. The factorial is obvious, the Fibonacci numbers only slightly less so. We need to keep track of the previous two Fibonacci numbers, as here:

```
DECLARE FUNCTION Fib% (n%)

FUNCTION Fib% (n)

' LOCAL variables are: I,First,Second,CurrentFib

IF n <= 1 THEN
 Fib% = n
 EXIT FUNCTION
ELSE
  First = 0
  Second = 1
   FOR I = 2 TO n
    CurrentFib = First + Second
    First = Second
    Second = CurrentFib
   NEXT I
 Fib% = CurrentFib
END IF

END FUNCTION
```

Although you can use the TIMER command to demonstrate the difference between the two versions, a more graphic demonstration is gotten by drawing a diagram of how much wasted effort there is in the recursive version of Fibonacci. Note that to compute FIB(5) recursively, a program has to compute FIB(3) twice, FIB(2) three times, and get to the trivial case 5 times. It never saves the information it so laboriously computes—it just recomputes it constantly.

Another example of where you can replace recursion are the procedures to LoadATree and SearchATree. Since the recursive call happens only at the end,

[7]in "Algorithms + Data Structures = Programs" p130 (italics in the original).

when a hole has not yet been detected, you may rewrite that part of the procedure in, for example, LoadATree, as here:

```
DO UNTIL WhereToGo% = -1

 IF Item$  < Nodes$(CurrentPos%) THEN
   WhereToGo% = Arrows%(CurrentPos%,0)
   Direction$ = "left"
 ELSE
   WhereToGo% = Arrows%(CurrentPos%,1)
   Direction$ = "right"
 END IF

LOOP
```

For those who are interested, the reason it's easy to change the original version to a non-recursive, iterative version is that LoadATree is a tail recursive program. This means that when you make the recursive call nothing is left to be done on the stack—it doesn't need to remember anything. Any tail recursive program can be changed to a loop, controlled by the trivial case—exactly as was done above. On the other hand, TreeSort is not tail recursive, because the stack contains many requests to PRINT a NODE.

One other point that may be of interest to you is that you can theoretically translate any recursive procedure, tail recursive or not, into an iterative version. The trouble is that you do it by setting up a stack and keeping track of everything—work best done by the compiler.

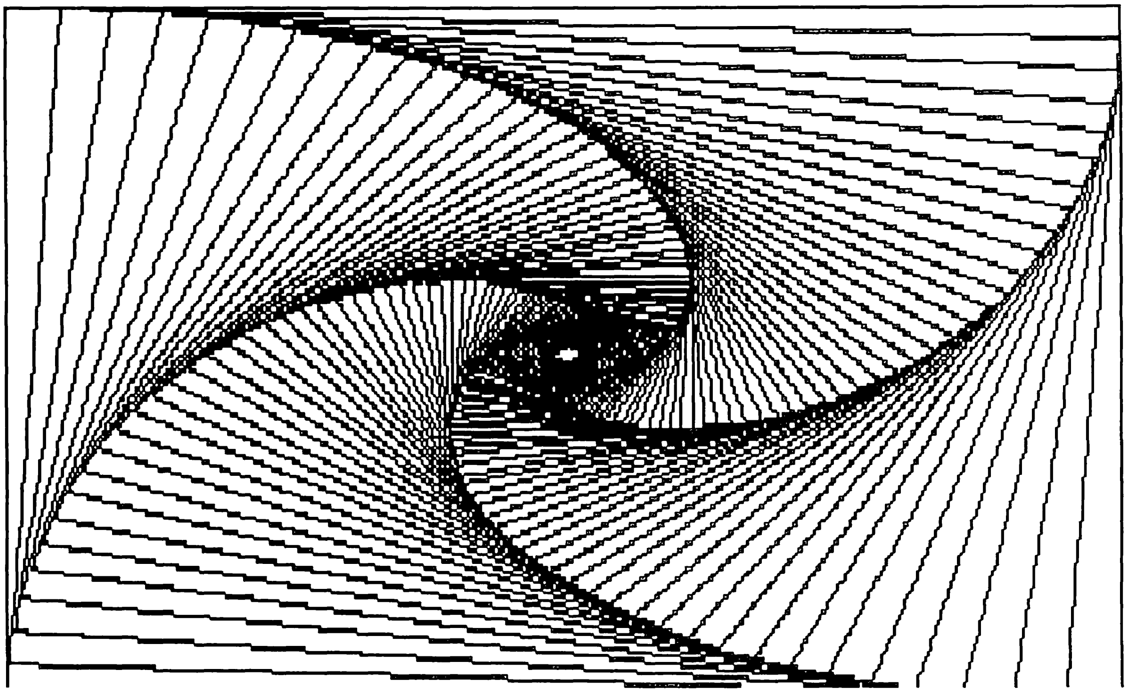
8 CHAPTER

Sight and sound: An introduction to graphics

Most of this chapter is about graphics—picture drawing by a computer. With suitable hardware, the graphics power of PCs, as released by QuickBasic, is astounding. Figure 8-1 is an example of what you can draw with a short 24 line program from the next chapter. Admittedly it will be a bit subtle, but still it's only 24 lines.

I should mention that although graphics are usually distinguished from text, this distinction isn't absolute. You can combine the LOCATE command with the CHR$ command to draw primitive kinds of pictures. In any case, the graphics statements in this chapter are far more powerful. They allow you, assuming you have suitable hardware, to control each dot, usually called a *pixel* or picture element, that appears on the screen. If you take a magnifying glass to the screen, then you can see that each character is made up of many of these pixels—but now you potentially can control each one.

In fact, with certain (more expensive, alas) combinations of hardware and



8-1 Nested squares.

monitors, you can divide the screen into 307,200 dots, and choose any one of 16 colors out of more than 200,000 to color them.

You should be aware that the on-line help files for the graphics commands are particularly good. It's worth consulting them, as well as chapter 5 of the Programming in BASIC manual, if you need more information on any of the commands I cover here. The manual also covers a few topics that I'm not covering here.

Computer graphics is a subject where advanced mathematics must inevitably rear its head. However, this chapter uses almost none. The next chapter on the other hand, including the example programs that drew the figure given above, will. In no case do I go beyond trigonometry. Of course, you can just skip the math and use the programs—the results are pretty spectacular.

Finally, I end this chapter with a short section on sound. Unfortunately, while QB's commands for sound effects are, in theory, quite powerful, in practice they are limited by the tinny speaker that most PCs have. For this reason, I don't think it's worth spending much time on them. Consult the Programming in BASIC manual for more details, if you disagree.

Getting started

You can only use QuickBASIC's graphics statements if you have the appropriate hardware, usually called a graphics board. QB allows you to control either a CGA (color graphics adapter), an EGA (enhanced graphics adapter) or the VGA that comes standard in the PS/2 line. To be precise, the PS2/25 and 30 almost have a VGA adapter. IBM calls it an MCGA and it's slightly less powerful. QB also supports the Hercules graphics and ATT's proprietary graphics card. Both of these, unlike the CGA, EGA and VGA, allow you to draw in only two colors, usually green and white. If you have a Herc card then you have to run a program called HERC.COM before running QB. The setup program would detect this for you.

Having the right kind of board is not enough; you also need the appropriate monitor. There are four kinds of monitors you can use. Monochrome monitors, which, in spite of the name use two colors, and three kinds of color monitors. To get graphics from a monochrome monitor one usually uses a Hercules or Hercules compatible board, but they will work with EGA and VGA boards as well. As for color monitors, the most primitive and the cheapest are called composite monitors. These are usually attached to a color graphics adapter (CGA). TV sets equipped with an RF modulator work like composite monitors.

Next up on the scale is an RGB monitor, which can be attached to all three types of boards. However, to take full advantage of a VGA card you need a special analog monitor. These start at around $300, and the price quickly escalates from there. A good way to think about an analog monitor is that with one you can easily adjust the exact amount of red, blue, or green in the signal. Analog monitors can be expensive—a 19 inch analog color monitor that can take full advantage of a VGA graphics adapter will cost more than a cheap 386sx computer!

Because the commands that control a color graphics adapter (CGA) will work with any color card, I'll use them to illustrate most of the ideas in this chapter. The results may be slightly different with the varying combinations of different hardware, of course. EGA and VGA boards recognize some specialized commands that I'll describe in section 5.

I should point out that it's possible to write a program that determines what kind of graphics board a computer has, and adjusts itself accordingly. You'll see one way to do this in the next to last section of this chapter.

The two graphics modes that work with all graphic boards are usually called medium resolution and high resolution. Medium resolution allows up to four colors on the screen, high resolution only allows two. The tradeoff is, of course, that the blocks (pixels) are much larger in medium resolution than in high resolution mode: the pictures are much finer in the latter.

Medium resolution divides the screen into 64,000 tiny rectangles, and you can control each one individually. After the command SCREEN 1, which clears the screen and enables (turns on) medium resolution graphics, QB sets aside a grid that's 320 columns across and 200 rows down. If you have a composite monitor it's better to say SCREEN 1,0. If you don't have a graphics adapter, then you'll get an error message. You identify each of the 64,000 blocks by means of its coordinates. Unlike with text (the LOCATE command), you place the column first and then the row. Moreover, both indices start with 0—so the columns are numbered from 0 to 319, and the rows from 0 to 199. For example,

(0,0)	is the top left corner
(319,0)	is the top right corner
(0,199)	is the bottom left corner
(319,199)	is the bottom right corner
(160,100)	is roughly in the center.

If two points have the same first coordinate then they're on the same vertical line; if they have the same second coordinate, then they're on the same horizontal line.

The next step is to decide what colors you want. This is done via the COLOR command, which takes two integer arguments.

```
COLOR X,Y
```

The first position specifies the background color, and here you have the most leeway. You can choose any one of 16 colors for this, and they are the same as for character (text) graphics. But, if you have an EGA or VGA, then section 5 will show you how to change these colors.

Here's the list:

0	Black	5	magenta	10	light green
1	blue	6	brown	11	light cyan
2	green	7	white	12	light red
3	cyan	8	gray	13	light magenta
4	red	9	light blue	14	yellow
				15	high intensity white

See the on line help for how these are interpreted on monochrome monitors.

Issuing a COLOR command blanks the screen and switches the background color, as the following fragment demonstrates:

```
SCREEN 1
LOCATE 1,1
PRINT "PRESS ANY KEY FOR THE NEXT COLOR"

 FOR CNumber = 0 TO 15
   LOCATE 12,1
   PRINT "This is color #";CNumber
   COLOR CNumber,0
   SLEEP
NEXT CNumber
```

The second entry, for example the 0 in the above fragment, in the COLOR command is a bit less powerful. This controls what is usually called the palette. And, just like an artist's palette hold the colors they have available, this second entry controls what colors you have available for the foreground. Also, because the

COLOR command blanks the screen, you can't use this command to change colors in midstream. This is only possible when you have an EGA or VGA board—the commands needed are described in section 5. Moreover, unless you have an EGA or VGA card, you're stuck with the standard colors. These are:

For Palate 0 (COLOR bkgrnd,1)

green
red
brown

For palette 1 (COLOR bkgrnd,2)

cyan
magenta
white

For example, the command COLOR 5,0 means that you have magenta, the background color, and green, red, and brown, the colors in the 0th palette available. Similarly, the command COLOR 14,1 gives you yellow, the background color, and cyan, magenta, and white. The colors contained in a palette are usually called the attributes of that palette. Think of them as the tubes of color that you have available for that particular palette.

To summarize: to get started in medium resolution graphics, issue two commands:

```
SCREEN 1,0
COLOR background,palette number
```

Even integers in the second entry give palette zero, and odd numbers give palette one.

When you are in medium resolution mode (SCREEN 1) you can easily mix text and graphics, although you do not normally have the higher order ASCII characters available. Versions of DOS after 3.1 have a means of doing this—check the DOS manual. As before, you use the LOCATE command to position the text—but you have to convert the text rows and columns in the LOCATE command to pixel coordinates to figure out where the text will appear. To calculate in which graphics rows and columns letters will appear, multiply the text row and column by 8 and reverse the positions. For example here:

```
SCREEN 1
COLOR 3,0
LOCATE 5,9
PRINT "Hello world!"
```

This puts you in medium resolution, turns the background color to cyan, and prints the message in brown letters starting at position (72,40). Text always appears in the third color of the current palette.

One important difference worth noting, as you saw in the demo program above, is that after a SCREEN 1 command, text appears twice as wide—40 characters to a line, instead of 80. These wide characters can also be used in text mode by issuing the command WIDTH 40. The command WIDTH 80 brings things back to normal. Both WIDTH commands erase the screen, so that you can't combine the two sizes.

In fact, the command SCREEN 0 which erases the screen and seems to bring you back to normal TEXT mode doesn't quite: it keeps the WIDTH at 40 and so the number of characters is 40/line. To completely return to normal, you need to issue two commands:

```
SCREEN 0
WIDTH 80
```

High resolution graphics are turned on by the command SCREEN 2. This erases the screen and sets up a grid 640 across and still 200 down. The row positions remain at 0 to 199, and the column positions are now numbered 0 to 639. If you have a Hercules card then SCREEN 3 gives you a 720 by 348 grid, but for simplicity I'll assume your resolution is 640×200. So for simplicity: (0,0) remains the coordinates of the top left corner, but (320,100) is roughly the center, and (639,199) is the bottom right corner.

In SCREEN 2 (and SCREEN 3)—high resolution—you only have two colors available, and text appears in its normal WIDTH 80 size.

Notice that 80 = 640/8 and 40 = 320/8 are the WIDTHs in the two modes. This isn't a coincidence. In both cases TEXT is made by filling in pixels in an 8×8 grid—only the size (width) of the pixels changes. The number of up-down pixels is always 8, so you don't double the 25 rows for text (=200/8) allowed per character.

Pixel control

O.K., now you know how the screen is divided in the two basic graphics modes. How do you turn a pixel on? The command is

```
PSET( , )
```

All you need to do is fill in the two entries, the first with the column and the second with the row. After QB processes this statement, then the pixel defined by that point lights up. Obviously where that point is depends on whether you've previously issued a SCREEN 1 or SCREEN 2 command: So after a SCREEN 1 command:

```
PSET(319,0)
```

would turn on the top right corner pixel, but after a SCREEN 2 command it would be in the center of the first row. The command PRESET(,) turns off the point.

It's possible to PSET outside the limits of the screen, for example: PSET(2000,1000)—no error message results. This is an example of *clipping*, which more generally refers to QB cutting off any part of an object that is off the screen. For example, the following simple program uses PSET to draw a straight line in medium resolution mode down the center of the screen, with a small amount of clipping:

```
'CH8\P1.BAS
' line via PSET with a bit of `clipping'

SCREEN 1
COLOR 1,0

FOR I =0 TO 250
 PSET (160,I)
NEXT I

END
```

Of course, when this program ends, you're still in graphics mode. To leave it and return to normal issue the SCREEN 0: WIDTH 80 combination, using the Immediate window.

The color used by PSET is usually the one with the highest number in the current palette. In high resolution, 0 is the background, and 1 is the foreground. You can change this in SCREEN 1. For example, in the fragment above the command COLOR 0,1 said I'm using the blue as the background and the 0th palette. Thus because yellow has the highest number in that palette, you'll get a yellow line. You choose among the colors in the current palette by a modification of the PSET command. For example, if you wanted to get a red line, color 2 of the current palette, use PSET(160,I), 2 in the fragment. Color 1 would match the background color and so be invisible.

In general the PSET command looks like:

```
PSET( column, row) , color code:
```

Suppose you wanted to erase every other dot in this line. Although there are many ways to do this, at this point the simplest is to notice that redrawing a point in the background color obviously erases it:

```
FOR I =0 TO 199 STEP 2
    PSET (160,I), 0
NEXT I
```

Another possibility is a variant on the PSET command, PRESET. PRESET(,) turns off the color of the pixel given in the parentheses.

Obviously, if you had to draw everything by plotting individual points, graphics programming would be too time consuming. Instead, QB comes with a rich supply of graphics commands, in computerese, graphics primitives, that allow you to plot such geometric figures as lines, boxes, circles, ellipses, or wedges, with a single statement.

For example, the statement:

```
LINE (160,0) – (160,199)
```

draws the line given in the first fragment given above. More generally, the statement:

```
LINE (StartColumn,StartRow) – (EndCol,EndRow),ColorCode
```

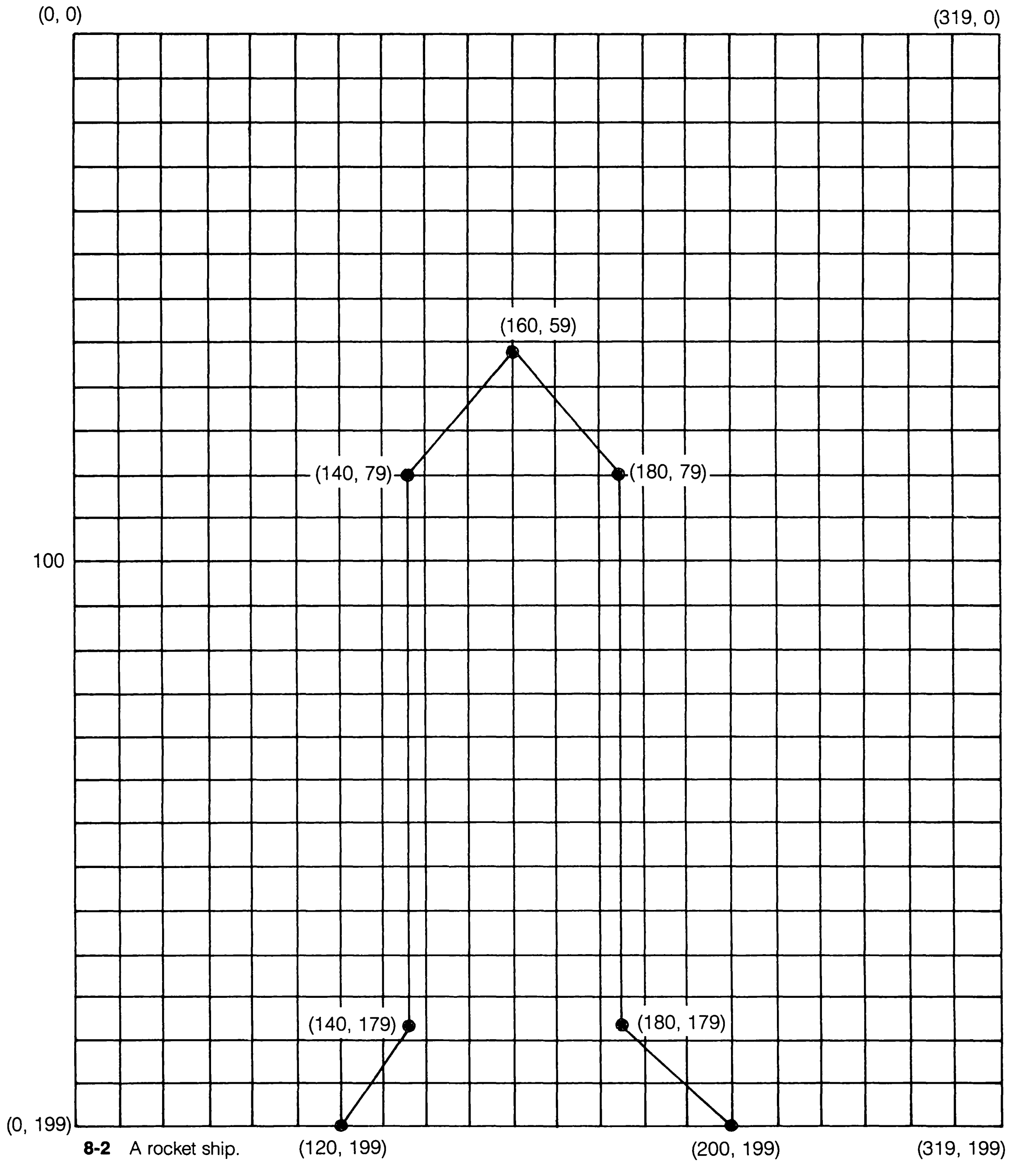
gives you a line connecting the two points with the given coordinates, using the color specified by ColorCode. For example, the next program gives you a starburst by drawing random lines in random colors from the center of the screen:

```
'CH8\P3.BAS
'random lines in random colors

RANDOMIZE TIMER
SCREEN 1,0

FOR I = 1 TO 100
 Col = INT(320*RND(1))
 Row = INT(200*RND(1))
 CCode = 1 + INT(3*RND(1))
 LINE (160,100) - (Col,Row) , CCode
NEXT I

LOCATE 24,1
PRINT "Press any key to erase screen";
A$ = INPUT$(1)
SCREEN 0
WIDTH 80

END.
```

The body of the FOR-NEXT loop calculates a random point and color code on each pass. Next, it draws a line from the center of the screen to that point. The block following the loop lets you get back to normal, or use the SLEEP command. I usually have a block like this at the end of a graphics program that I'm developing. This is because the ubiquitous "Press any key to continue" that appears when a QB program finishes often wipes out a portion of the painted screen.

Suppose you wanted to draw a rocket ship as in the Fig. 8-2. Since you can read off the coordinates from the diagram it's easy, if a bit tedious, to write the program:

```
`CH8\P3.BAS
' A rocket ship

SCREEN 1

LINE (120, 199)-(200, 199)
LINE -(180, 179)
LINE -(180, 79)
LINE -(160, 59)
LINE -(140, 79)
LINE -(140, 179)
LINE -(120, 199)

SLEEP
END
```



8-2 A rocket ship.

Although it's possible to draw almost anything by outlining it using graph paper, just mimic what I did earlier. Obviously, as the object becomes more complicated, it becomes less and less practical. One of the reasons why mathematics is needed for computer graphics is to give formulas for various complicated objects. The formulas then shorten the length of the program because they themselves incorporate an enormous amount of information. This makes it practical to write the program whereas writing a few thousand PSET statements is obviously not.

A modification of the LINE command lets you draw a rectangle. The statement:

```
LINE (FirstCol,FirstRow) – (SecCol,SecRow),CCode,B
```

draws a rectangle in the given color code (CCode) whose opposite corners are given by (FirstCol,FirstRow) and (SecCol,SecRow). For example, the following fragment gives you "nested boxes" in the high resolution screen

```
SCREEN 2
    FOR I = 1 TO 65 STEP 5
        LINE (5*I,I) – (639 – 5*I,199 – I), ,B
    NEXT I
```

Notice that I've left off the color code, but still kept the comma to separate out the B. Without this comma, QB would think the B was the name of a variable, rather than the box command. Leave out the comma and QB would think you're asking for a line connecting (5*I,I) – (639 – 5*I,199–I), with color code the current value of B (probably 0, so you'd get nothing!). I also could have put a 1 for the color code.

Say BF rather than B and you get a "filled" box, so:

```
LINE (FirstCol,FirstRow) – (SecCol,SecRow),CCode,BF
```

yields a solid rectangle whose opposite corners are given by (FirstCol,FirstRow) and (SecCol,SecRow). For example, change the fragment to read:

```
DEFINT A-Z
SCREEN 1
COLOR 0,0
    FOR I = 1 TO 32 STEP 3
        CCode = I MOD 4
        LINE (5*I,I) – (320 – 5*I,199 – I),CCode,BF
    NEXT I
```

and you get a rather dramatic nesting of colored frames. This happens for two reasons. The first is that the MOD function lets you cycle through the color codes in order, and the second is that when QB draws each smaller rectangle it overdraws part of the previous one, using the new color.

Suppose you wanted to place a square on the screen. Your first instinct might be to use (say in high resolution SCREEN 2) 3.2 columns for each row, because there are 640 columns and only 200 rows. This may work, but probably won't—because monitor screens are rarely square. You are likely to run into what is usually called by the forbidding term of the problem of aspect ratio. *Aspect ratio* is simply the ratio between the distance between two pixels in adjacent rows and columns. If the distance between each column is exactly the same as the distance between each row then the aspect ratio is one and you have the ideal situation. It's easy to draw a square because the fudge factor will be 3.2 columns/row in SCREEN 2, and 1.6 in SCREEN 1. If not, then you have to calculate the fudge factor. The best way to do this is to check your monitor with a little test program that I'll show you next. A good bet, however, is that on most monitors the height of the screen is 3/4 of the width. So use 2.4 (=3.2*3/4) columns for each row in high resolution, and 1.2 columns/row in medium resolution in order to get close to a perfect square.

The following is the test program. Run it and keep track of when the rectangle appears squarest—that's the number of columns to use per row in high resolution, half that number in medium resolution:

```
'CH8\P5.BAS
' aspect ratio test

SCREEN 2

Aspect = 1.5

LOCATE 1,5
PRINT "This program demonstrates aspect ratios for the high";
PRINT " resolution"
PRINT "screen.  It starts at 1.5 columns/row and moves by .1";
PRINT " increments"
PRINT "to 3 columns/row."

PRINT:PRINT  "Press X to end, any other key to continue."
A$ = INPUT$(1)
CLS

DO Until Aspect >= 3 OR A$ = "X"
 LINE(150,50) - (150 + 100*Aspect,150),,BF
 LOCATE 24,1
 PRINT USING "This is aspect ratio #.#";Aspect;
 PRINT ".  Press X to end, any other key to continue.";
 A$ = INPUT$(1)
 Aspect = Aspect + .1
LOOP

END
```

Although QB has many powerful tools to do animation (see section 5.1 of the programming in BASIC manual), you now have the tools to simulate one kind: a so called drunkard's walk, or a random walk as it's more technically called. All this means is that one imagines an object whose movements over time we can plot. If it seems to move randomly (like some people said the stock market does—or did until Oct. 19, 1987), then you have a random walk. To simulate this, I'll put a tiny square in the middle of the screen and move it around, erasing the previous square each time. However, instead of moving the square a fixed amount, I'll have it move up and down and left and right randomly. As you'll see, the square will spend most of its time in a narrow range around the center:

```
'CH8\P5.BAS
' moving squares to imitate a `random walk'

RANDOMIZE TIMER
SCREEN 1
COLOR 0
DEFINT A-Z
X = 160: Y = 100

FOR I = 0 TO 500

 XMove = 6*RND(1)
 YMove = 5*RND(1)

 IF RND(1) < .5 THEN X = X + XMOVE ELSE X = X - XMOVE       'A
 IF RND(1) < .5 THEN Y = Y + YMOVE ELSE Y = Y - YMOVE

 IF X < 0 OR X > 293 OR Y < 0 OR Y > 194 THEN
  ' DO NOTHING
 ELSE
  LINE (X,Y)-(X + 6,Y+5),2,BF
  LINE (X,Y)-(X + 6,Y+5),0,BF                               'B
 END IF

NEXT I

LOCATE 24,1
PRINT "Press any key to return to text mode";
A$ = INPUT$(1)
SCREEN 0
WIDTH 80

END
```

A) This gives the random motion: the box moves up or down and left or right, depending on whether the random number generator delivers a number less than a half or not.

B) By redrawing the rectangle in the background color it's erased, giving the animation. If you remove this, then, the animation is lost, but it's replaced by a visible trace of where the box has been. Change the ELSE clause to read:

```
CCode = 1 + 3*RND(1)
LINE (X,Y)-(X + 6,Y+5),CCode,BF
```

and the results are usually a quite attractive random pattern.

QB keeps track of where it stopped plotting. This is usually called the last point referenced (LPR). If you are continuing a line from the last point referenced, QB allows you to omit it in the LINE command. For example:

```
LINE - (160,90)
```

draws a line from the last point referenced to the point with coordinates 160, 90. When you start any graphics mode with a SCREEN command, then the last point referenced is the center of the screen. Moreover, the screen clears and the default color is white. After a LINE command, the last point referenced is the end point of the line (the second coordinate pair).

Up to now you've been using absolute coordinates. Each point is associated with a unique row and column. It's occasionally useful to use relative coordinates where each point is defined by how far it's away from the last point referenced. For example, if you say:

```
PSET(12,100)  (or PRESET(12,100) )
```

which makes 12,100 the last point referenced, then you can say:

```
PSET STEP(50,10)
```

and you will have turned on the point in column 62 (50 + 12) and row 110 (10 + 100). In general when QB sees the command:

```
STEP (x,y)
```

in a graphics command it uses the point whose coordinates are x units to the right or left and y units up or down from the last point referenced, depending on whether x and y are positive or not.

Circles, ellipses and pie charts

Normally, to describe a circle in QB you give its center and radius. After say a SCREEN 1 command:

```
CIRCLE (160,100), 60
```

draws a circle of radius 60 in the third color of the current palette. The last point referenced after a CIRCLE command is always the center of the circle: (160,100) in the example above. On the other hand:

```
    CIRCLE (160,100),60,CCode
```

would draw a circle of radius 60 in the color code indicated here by the variable CCode. Next is a sample program that shows off this version of the circle command:

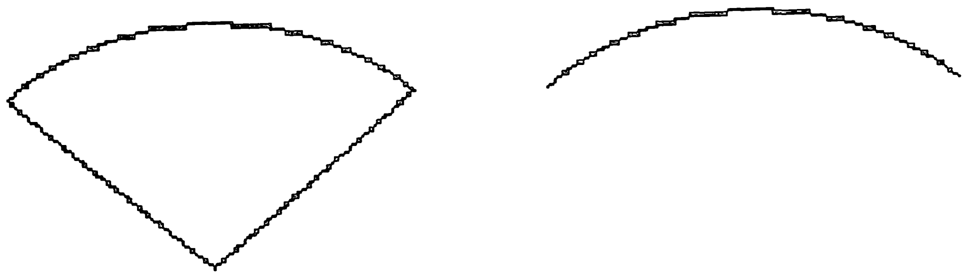
```
'CH8\P6.BAS
' nested circles

DEFINT A-Z
SCREEN 1
COLOR 1,1

FOR I = 59 TO 259 STEP 4
  CCode = 1 + (I MOD 3)
  CIRCLE (I,100),60,CCode
NEXT I

SLEEP
SCREEN 0
WIDTH 80

END
```

Of course, these circles are a bit jagged, because the resolution in SCREEN 1 is minimal. If you don't mind the loss of color, then you can smooth them up considerably by using high resolution (SCREEN 2) graphics.

You may be wondering what the radius is exactly—is it 60 columns, or 60 rows, or both 60 columns and 60 rows, as a mathematical radius would be? It turns out that the CIRCLE command usually counts pixels by columns to determine the radius. It then scales the number of rows by dividing by 1.2. So a circle of width 60 would take 60 columns and only 50 rows, in medium resolution mode. In high resolution it divides the number of columns by 2.4 to get the number of rows. So the CIRCLE command in this program had QB plot what it hopes will be a circle by assuming that your monitor has the width to height (4/3) ratio discussed earlier. Obviously, if you have a monitor that has a different aspect ratio, then you must override QB's assumption. This is done by a variant on the CIRCLE command that you'll see shortly.

You may have seen pie charts used to display data. QB sets up a pie chart with a modification of the circle command. First some terminology: a sector is a pie shaped region of a circle, and an arc is the outer boundary of a sector. See Fig. 8-3.

To draw a sector or arc you have to tell QB what angle to start at, and at which



8-3 A sector versus an arc.

angle to finish. This is done using radian measure, which you may have seen in school, and is used in the trigonometric functions in QB. Radian measure isn't very difficult: it measures angles by what percentage of the circumference of a circle of radius one that it gives. All the way around is called 2π radians because 2π is the length of the circumference of this circle. So 360° is approximately 6.29 radians, 1/2 a circle is π radians (180°), 1/4 circle is $\pi/2$ radians (90°), and so on. To go from degrees to radians, multiply by $\pi/180$, to go back multiply by $180/\pi$. In any case, the statement:

```
CIRCLE (XRad,Yrad), radius,CCode, StartAngle,EndAngle
```

draws an arc of the circle starting at the angle, indicated in radians, by StartAngle, and ending it with Endangle. To get a sector, use negative signs. So, assuming that you're in SCREEN 2 and have set up a variable called Pie = 3.14159 (= 4*ATN(1) = π), then:

```
CIRCLE (150,100),160,1,-Pie/4,-3*Pie/4
```

gives you the sector in Fig. 8-3. And:

```
CIRCLE (150,100),160,1,Pie/4,3*Pie/4
```

gives you the arc.

There are a few peculiarities of these commands that I should mention: the first is that, although mathematics allows negative angles, QB does not. The negative sign only serves to indicate: "Draw a sector rather than an arc." The second is that if you want your arc to start with a vertical line pointed due east, i.e., 0 degrees = 0 radians, you shouldn't use −0 for the StartAngle or EndAngle. Instead use $-2*\pi$ (= −6.28 . . .). The final peculiarity is that angles in the CIRCLE command can only have values between -2π (−6.28 . . .) and 2π (6.28 . .).

Suppose we wanted to write a general pie making program. By that I mean a program that takes a bunch of numbers stored in an array, and sets up a pie chart using the numbers. Essentially, what we need to do is to determine what percentage of the total each positive number is, and set up an arc using that percentage. Here is the procedure:

```
SUB MakePie( A(),SizeOfCircle)

' This procedure takes an array of positive entries and
' creates a pie chart using proportions determined by the
' array.
' LOCAL variables: I,First,Last,Total,StartAngle,EndAngle

SHARED TwoPie                           'TwoPie should be 6.28...
DIM I, First, Last AS INTEGER                '8*ATN(1)

First = LBOUND(A,1)
Last = UBOUND(A,1)
Total = 0

FOR I = First TO Last
 Total = Total + A(I)
NEXT I

SCREEN 2

StartAngle = -TwoPie

FOR I = First TO Last
 EndAngle = ((A(I)/Total)*TwoPie) + StartAngle       'A
 CIRCLE (320,100),SizeOfCircle,1,StartAngle,EndAngle
 StartAngle = EndAngle
NEXT I

END SUB
```

A) This is the key. A(I)/Total gives what fraction of the total a particular entry is. Multiplying by TwoPie (=2*3.14159 . .) gives you the radian equivalent. Since the StartAngle is $-2*\pi$, adding this angle gives you the necessary negative number for the size of the sector, starting due east and going counterclockwise.

How can you test this procedure? Simply create some random arrays, of random sizes, with random positive entries, and CALL the procedure.

The CIRCLE command allows you to adjust the aspect ratio by adding one more option:

```
CIRCLE [STEP]  (XCenter,YCenter),radius,,,,aspect
```

The four commas must be there, even if you are not using the color code and angle options that you saw earlier (STEP is optional, of course). This version of the CIRCLE command lets you change the default ratio of columns to rows. It's really an ELLIPSE command.

If the aspect ratio is less than one, then the radius is taken in the column direction, and the ellipse is stretched in the horizontal direction. If the aspect ratio is greater than one, then the radius is taken in the row direction, and the ellipse is stretched in the vertical. Here is a program that demonstrates this:

```
'CH8\P7.BAS
' Aspect ratio test for ellipses

SCREEN 2
PRINT "Press any key to continue"
SLEEP

FOR I = .1 TO 2 STEP .1
 CIRCLE (320, 100), 75, , , , I
 LOCATE 24, 10
 PRINT "This is aspect ratio"; I;
 PRINT " Press any key to continue";
 SLEEP
 CLS
NEXT I

SLEEP

END
```

Although this program stops when the aspect ratio reaches 2, you could continue making it bigger. As the aspect ratio gets larger, the ellipses get closer and closer to a vertical line.

You also can fill in enclosed areas using the PAINT command. The command:

```
PAINT [STEP] x,y,color,bordercolor
```

begins filling in the area, and stops whenever it gets to a pixel of the color given by bordercolor. If you leave this out, then QB stops painting when it encounters a pixel whose color is that of the paint command itself, the third entry. And, as always, the STEP is optional.

How the paint command will actually work in practice is a bit subtle, but a good analogy to keep in mind is a spreading paint slick. A slick will stop whenever it hits a boundary point. On the other hand, it could continue forever if there's a hole in the boundary, however small. The following two fragments show this. First, you can paint a circle:

```
SCREEN 1
CIRCLE (160,100),75
PAINT (160,100)
```

This gives you a white circle. Now insert a PRESET command to turn one pixel off:

```
    SCREEN 1
    CIRCLE (160,100),75
    PRESET (235,100)
    PAINT (160,100)
```

and rerun the fragment. What you'll see is the whole screen gradually turning white. The moral is, make sure your boundaries are solid. I should point out the whole concept of boundary is a subtle one in mathematics, as is finding an inside point. In this case the only points on the boundary that are easy to calculate are the ones on the horizontal.

Usually lines are solid and paint fills with a solid color, but it's possible to change this. See sections 5.3.2.3 and 5.8.2 of the Programming in BASIC manual.

The Draw command

The rocket ship program was short but tedious. The DRAW command gives you a much more powerful way to create any graphics figure that you can sketch. Essentially the DRAW command gives you control of a pen that you move and lift as needed. Moreover, once an object is drawn you can store it, rotate it, scale it, or move it with a single command.

For example, the command DRAW "U10" draws a 10 pixel line directly up from the last point referenced. In general, the DRAW command uses certain combinations of command strings and numerals. Here are a few more of these command strings:

U	up
D	down
L	left
R	right
E	diagonally up and to the right
H	diagonally up and to the left
F	diagonally down and to the right
G	diagonally down and to the left

These commands alone are enough to let you imitate the famous Etch A Sketch toy, and this is a standard, but still fun, programming problem. For another example, the following fragment redraws the rocket ship given before. First we make (120,199) the last point referenced:

```
    PSET (120,199)          'There are other ways to do this
```

Now enter:

```
DRAW "R80  H20  U100  H20  G20  D100  G20"
```

Across 80 pixels, then up diagonally 20 pixels, then straight up 100 pixels, etc. Here, as in any DRAW statement, the spaces are there to improve readability; QB doesn't care.

However, you can do much more. Set up a string:

```
RocketShip$ = "R80  H20  U100  H20  G20  D100  G20"
```

Now to have two rocket ships on the screen:

```
PSET (50,199)          'set LPR
Draw RocketShip$
PSET (200,199)         'reset LPR
Draw RocketShip$
```

Most of the power of the DRAW command comes after you've set up string variables that will draw the figures that you're interested in. This is because you can add commands before you issue a DRAW command, or inside a DRAW command, that scale, rotate, change color, or paint the object. For example:

DRAW "Snumber"	Enlarges the object by a factor given by the value of the number divided by four.

This means that the command DRAW "S4" gives you the size indicated in the original string. The number must be between 1 and 255. See below for a way to incorporate a numeric variable.

Scaling can easily lead to clipping your image. The command DRAW "S64" enlarges your figure by a factor of 16—very few figures will fit after this.

For an example of scaling, try the following:

```
SCREEN 1
PSET (20,199)
DRAW "S1"
Draw Rocket$
```

Now try the following program:

```
'CH8\P8.BAS
' An armada

CLS
ROCKET$ = "R80 H20 U100 H20 G20 D100 G20"
SCREEN 1
```

```
LOCATE 1, 10
PRINT "Buck Rogers redeux"

FOR I = 1 TO 4

  SCALE$ = "S" + STR$(I)                          'A
  DRAW SCALE$
  NewPos = NewPos + 5 + (20 * I)                  'B
  PSET (NewPos, 199)
  DRAW ROCKET$

NEXT I

SLEEP

END
```

A) This is the most interesting line. Although there are other ways to incorporate numeric variables into a DRAW command, this is by far the simplest: change the numeric variable using STR$ and concatenate with a plus sign.
B) The position of the next rocket has to take into account the size of the rocket. Scale 1 is one quarter size, S2 half size, etc. The rockets would be 20,40,60 and 80 pixels across.

Now let's go back to the analogy of a pen plotting. The prefix command B lifts the pen. So the command:

```
DRAW "BU10"
```

moves the last point referenced 10 points up, but does no plotting. The command:

```
DRAW "M x, y"
```

moves to the point x,y. If the pen is down, this has the same effect as: LINE − (x,y). For example, when the pen is down the command DRAW "M +x, +y" would work like:

```
LINE – STEP (X,Y)
```

A plus or minus in the M command means uses relative coordinates. On the other hand:

```
DRAW "BM x,y"
```

is a rapid way to move the last point referenced without plotting the point x,y. For example, you could use it to remove the PSET commands from the previous program.

The prefix N allows you to plot without changing the last point referenced:

```
DRAW "ND10"
```

draws a line 10 pixels long, but doesn't change the LPR.

Suppose we wanted to make the rocket ship move. One way to do this is to:

DRAW Rocket$
Change to the background color, wait a bit
Draw Rocket$ again
Change the LPR
reDRAW the rocket

The command string to set the drawing color is C. The following is a program that moves the rocket. The conversion of the outline is complicated by the fact that a statement like SLEEP .1 isn't yet possible:

```
'CH8\P9.BAS
'another way to animate

DECLARE SUB Delay (x!)

SCREEN 1
Rocket$ = "R80 H20 U100 H20 G20 D100 G20"


FOR I = 199 TO 50 STEP -10
    Row$ = STR$(I)
    DRAW "s1"
    DRAW "BM" + "140" + "," + Row$                  'A
    DRAW Rocket$
    CALL Delay(.1)
    DRAW "C0"
    DRAW Rocket$
    DRAW "C3"
NEXT I

SUB Delay (x!)

StartTime = TIMER

  DO UNTIL TIMER - StartTime >= x
   LOOP

END SUB
```

A) This again creates a string using concatenation. The DRAW command also lets you rotate a figure a certain number of degrees (not radians!). The prefix TA, followed by the number of degrees, rotates the figure to be plotted. For example:

```
DRAW "BM 100,160"
DRAW "TA 45"
DRAW Rocket$:
```

gives you the original sized rocket at a 45 ° angle, counterclockwise.

Finally, the command DRAW "P x,y" works exactly like the command PAINT (x,y), using whatever color has been set by the previous C command.

```
DRAW "TApaint
```

Other SCREEN modes —some powers of EGA and VGA cards

EGA and VGA cards give you enormous powers. The help files that explain how to control them are about 20 pages long, and so they're a bit intimidating. What I want to do in this section is to give you an introduction. Hopefully, after reading it the help files won't be quite so intimidating.

So far you've seen SCREEN 1 and 2. As I mentioned, SCREEN 3 for Hercules cards works like SCREEN 2, except the resolution is higher. Here's a brief explanation of what the other screen modes can do:

SCREEN 4: Is for ATT and Olivetti computers only. This isn't documented in the manual. I found it out by looking at the help file, and I haven't got one of those machines to check out what follows. In this mode you have: 640 columns, 400 rows, text is 80 characters/line, 16 colors through the COLOR statement—background is always black.

All the other screen modes require at least an EGA card.

SCREEN 7: EGA or VGA. 320×200 graphics—text is 40 characters with 25 lines. Can choose 16 foreground colors (any of 16 attributes) in a single palette, with 16 background colors.

SCREEN 8: EGA or VGA. 640×200 graphics—text is 80 characters with 25 lines. Can also choose 16 colors for any of 16 attributes in a single palette with 16 background colors.

SCREEN 9: EGA or VGA. 640×350 graphics. 80 characters for either 25 or 43 lines. Depending on the memory on your graphics card, you can choose either 4 or 16 colors from 64 for the foreground, and 64 colors for the background.

SCREEN 10: EGA or VGA with a monochrome monitor only. 640×350 graphics. 80 characters for either 25 or 43 lines.

SCREEN 11: VGA (or a PS/2/25-30's MCGA, the "semi VGA"), 640×480, 80 characters per line for either 30 or 60 lines. Two foreground colors chosen from among (honestly) 262,144 (256K) colors. Expect the differences to be subtle.

SCREEN 12: VGA only. 640×480, 80 characters per line for either 30 or 60 lines. This time there will be 16 foreground colors, chosen from the same 262,144 colors as in screen 11.

SCREEN 13: VGA (or the "semi VGA" of a PS/2/30). 320×200 resolution, 40 characters for only 25 lines. Allows up to 256 foreground colors, chosen from the same 262,144 colors.

Whenever you are allowed more than 25 lines, then the command WIDTH is used to do so. For example, WIDTH 43 sets the number of lines to 43, after a SCREEN 10 command.

Suppose for a second that you are in SCREEN 1. Recall that in SCREEN 1 you had two palettes, each with four fixed colors. These tubes of paint are, as I said earlier, usually called attributes. The numbers used in coloring pixels via any graphics command are therefore called attribute numbers.

Now suppose you have an EGA or VGA card. Then you can change these previously fixed colors—change the colors assigned to the different attribute numbers. For example, the command:

```
PALETTE 1,14
```

would change the color in attribute one to the color with code 14, which happens to be yellow. The 16 colors that you can use to replace the default ones are exactly those for the background colors mentioned earlier. A PALETTE command with no other entries restores the default colors.

In general, the PALETTE command chooses the colors for any of these modes by assigning a color number to an attribute—the number of attributes, of course, depending on the SCREEN mode. Attributes are then used in commands like DRAW "C_", LINE and CIRCLE to draw objects. For example, the third slot in the LINE command:

```
LINE (x1, y1 ) – (x2, y2),attribute,[B[F] ]
```

statement would give you a filled box in the color specified by the current value of attribute.

Moreover, the PALETTE command will change the screen instantaneously. If a part of the screen appears in attribute 2, and you were allowed say 64 colors, then the command PALETTE 2,53 would recolor that part of the screen in the 53rd color. Because there are 15 attributes possible in text mode when you have an EGA or VGA card, you also can use the PALETTE command to change the colors for TEXT (SCREEN 0) as well.

One place to show off the PALETTE command is in SCREEN 9. In this mode, if you have enough memory on your graphics board, then you can have 16 attributes (colors) on the screen at any one time, chosen from the 64 background colors.

For the following example program recall that text appears in the highest color in a given palette, and the background is in the lowest color of the palette.

The following is a program that shows off changing the text colors using the PALETTE command:

```
'CH8\P10.BAS
' A demonstration of the PALETTE command in SCREEN 9

DEFINT A-Z
SCREEN 9

FOR ColorNumber = 0 TO 63

  PALETTE 0, ColorNumber                        'A
  CLS
  TextColor = (ColorNumber + 1) MOD 64

  PALETTE 15, TextColor                         'B

  LOCATE 1, 1
  PRINT "The background color is"; ColorNumber;
  PRINT "while the text appears in color number"; TextColor
  SLEEP 2
NEXT ColorNumber

END
```

A) Since the PALETTE command works instantaneously, this changes the background color on each pass.
B) As I mentioned, text appears in the highest color of the palette. The program is always using the next highest numbered color for this. The MOD command is just to wrap around when it gets to 64.

Of course, a more dramatic demonstration would be to change all the colors in a palette simultaneously. This is done using the PALETTE USING statement. The command:

```
PALETTE USING array
```

where array is an array of integers or long integers, changes the colors in the palette by using the integers (long integers) stored in the array for the color attributes. You must DIMension the array to have at least as many entries as there are attributes. For example, suppose you have:

```
DIM A% (0 TO 31)
A% (0) = 12
A% (1) = 15
A% (2) = 37
.
.
.
```

then PALETTE USING A would change the first (0th) attribute to color number 12, the second to color number 15, and so on. If an entry is −1 in the array, then the attribute stays its original color. Any other negative entry in A, or one too large, would give an error message.

I dimensioned this list at 31 (=32 entries) to show off an optional feature of PALETTE USING. For example, I could say:

```
PALETTE USING A(16)
```

and now the colors would be taken starting from the 16 and going to the 31 entry. The general form is:

```
PALETTE USING arrayname(index)
```

where index determines where to start. You must have DIMensioned the array so that there's enough room starting from the index to fill all the attributes.

Next is a program that changes all 16 colors in SCREEN 9's palette simultaneously. Because you can only displace 16 colors at any one time, it shows them off in four groups (colors 0 −15, 16 −31 etc.).

```
DECLARE SUB Demo (A%(), I%)

' A demonstration of the PALETTE USING command in SCREEN 9

DEFINT A-Z
DIM Attrib(0 TO 63)

SCREEN 9

FOR I = 0 TO 3

 FOR J = 0 TO 15
  Attrib((16 * I) + J) = (16 * I) + J                    'A
 NEXT J

 CALL Demo(Attrib(), I)

NEXT I

END

' THis SUB demonstrates the PALETTE USING command

SUB Demo (A(), J)

'LOCAL Variable Count

CLS

PALETTE USING A(16 * J)                          'B

FOR Count = 0 TO 14                              'C
```

```
 LINE (0, 22 * Count)-(640, (22 * Count) + 21), Count, BF
NEXT Count

LINE (0, 330)-(640, 350), 15, BF

LOCATE 1, 10
PRINT "This shows off colors "; 16 * J; " to"; (16 * J) + 15;
PRINT " Press any key to continue.";
I$ = INPUT$(1)

END SUB
```

A) This fills an array with the integers from 0 to 63, in groups of 4, indexed by I. I can then use the counter I in the call to the subprogram.
B) This moves the starting index along for the PALETTE command. On the first call you start with 0, at the second with 16, and so on.
C) This draws colored boxes 22 lines wide, except for the last box which is only twenty lines wide.

Because the PALETTE USING A( ) command works instantaneously, I didn't really need to write the program this way. Do you see how it might be changed?

I mentioned that you might need a long integer array for the PALETTE USING command. This is because in SCREENs 11-13 the color numbers can be as large as 4,144,959. For the analog monitors that are needed in these modes, colors are determined by explicitly setting the intensity of the blue, green and red signal according to the formula:

(65536*Blue Intensity) + (256*Green Intensity) + Red Intensity

where the intensity for all the colors is a number between 0 (essentially don't use that color) and 63 (maximize that color component). A color code of:

(65536*63)

gives the bluest color possible, (256*63) the greenest, and 63 the reddest. 4144959 gives the one closest to white, and 0 is closest to black. By combining the amount of blue, green and red intensity you can customize colors to your heart's content.

Finally, the easiest way to determine what kind of video card a machine has is to use error trapping. The idea of this module is that you know that, for example, if SCREEN 12 is not allowed but screen 11 is, then you have an MCGA. Similarly, if SCREEN 7 is allowed but not screen 11 then you have an EGA, and so on.

The actual programming is a bit tricky. What I chose to do is set up an array for the possible modes. Next, I set up a loop which stops either when I've tried all the modes, or I managed to use the SCREEN statement without going to the error handler. The easiest way to do this is to use a variable that I'll call Found. In each

pass of the loop, Found is set back to TRUE but the error handler, if invoked, always switches it back to FALSE. This works because my loop is only tested at the top.
Here is this module:

```
'find out acceptable SCREEN modes via error handler

CONST FALSE = 0, TRUE = NOT (FALSE)
DEFINT A-Z
DIM Modes(1 TO 7)
ON ERROR GOTO VideoHandler

FOR I = 1 TO 7
  READ Modes(I)
NEXT I

ModesTried = 1
Found = FALSE

DO UNTIL ModesTried = 6 OR Found
    Found = TRUE
    SCREEN Modes(ModesTried)
LOOP

PRINT "The screen mode you can use is"; Modes(ModesTried)
END
DATA  12,11,7,4,3,2,0

VideoHandler:
  ModesTried = ModesTried + 1
  Found = FALSE
  RESUME NEXT
```

Sound

Sound is measured in Hertz (Hz is the abbreviation). 1 Hz would be 1 cycle per second. Ideally, but rarely in practice, humans can hear sounds between 20 Hz and 20,000 Hz. The BEEP command gives a sound at 800 Hz for about a quarter second. The SOUND command lets you, theoretically, play a given pure tone for a given amount of time. It's used as follows:

```
SOUND frequency in Hz, duration
```

The duration is measured in clock ticks, which are about 1/18 of a second (1/18.2 to be precise). So the command:

```
SOUND 800,18
```

is like BEEPing four times. The command SOUND 523,18 would play a sound close to a middle C (523.25 is the frequency of middle C), but frequencies in QB must be integers.

Combining the SOUND command with a FOR-NEXT loop gives you an easy way to incorporate sound effects into your programs:

```
FOR I = 100 TO 5000 STEP 10
    SOUND I,1
NEXT I
```

SOUNDs beyond 5000 Hz are unlikely to be audible on a PC's speaker.

QB can also use the PLAY command to generate music. To use this facility you have to know how music is scored. If you do, the on-line help for the PLAY command is quite detailed, and those who are interested might want to look there.

It's possible to write a program that can automate the process of entering notes, and anyone who reads the Help file and knows enough about music shouldn't have much of a problem doing so. But in all honesty IBM PC's are not known for their musical abilities.

As a sampling, however, of what you have to do, here is how to play something that sounds like the first few notes of "Auld Lang Syne":

```
PLAY "03  C8  F8.  F4  F4  A4  G4.  F4  G4  A4  F4.  F4  A4"
```

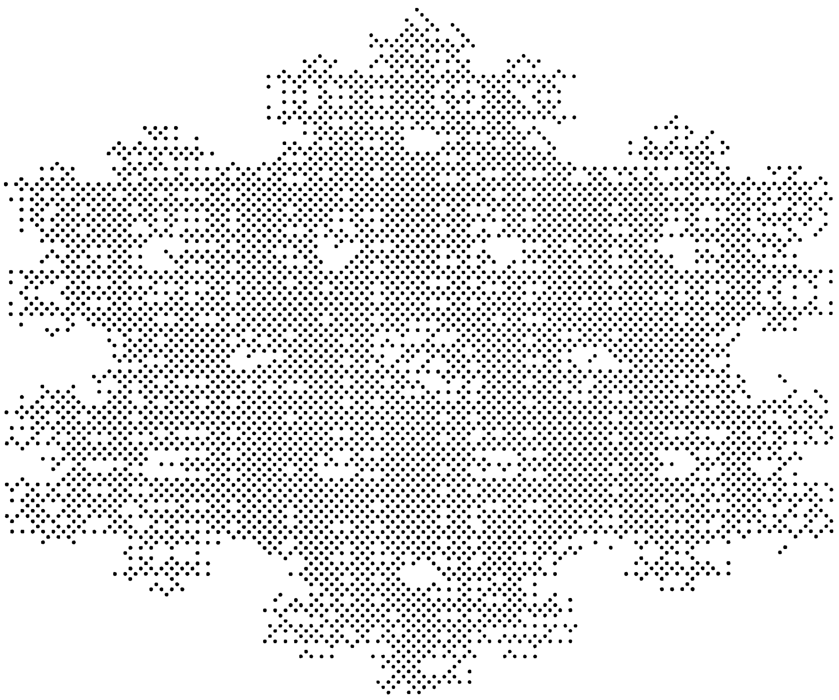
9 CHAPTER

Graphics: Curves and fractals

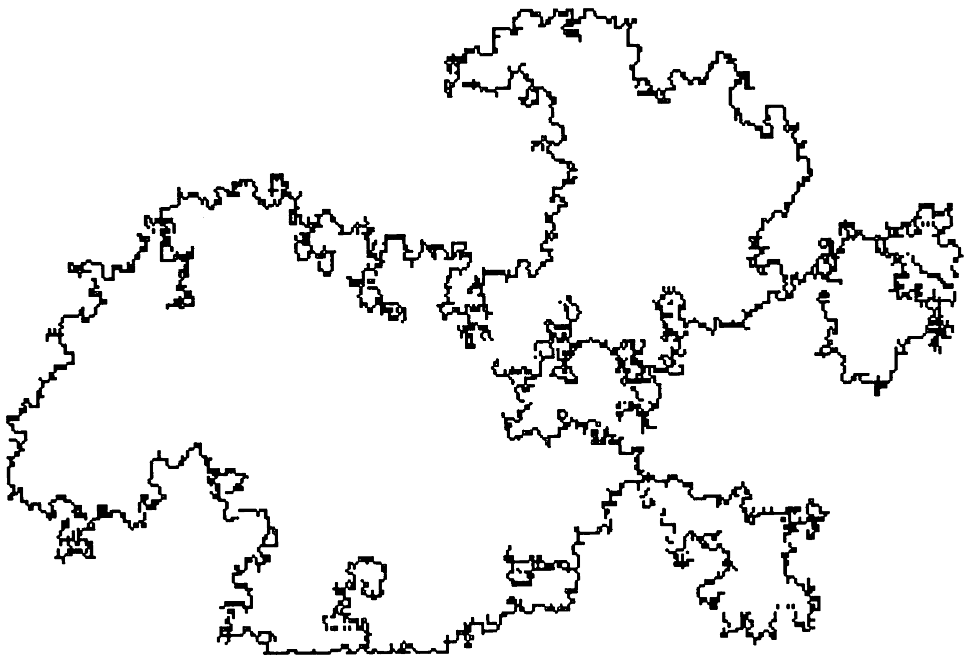
O.K, here's where the math starts. The first section uses the X-Y plane (Cartesian plane), and so a tiny bit of analytic geometry. Section 2 uses polar coordinates, and the last section uses some trigonometry. Fair warning.

But it's worth it. The last section of this chapter introduces you to fractals. These are amazing objects whose study continues to be at the forefront of current research in computer graphics. Figure 9-1 and Fig. 9-2 on page 242 are examples of what the programs described there can draw. As you can see, this is a reasonable portrait of what a coast line (two dimensional landscape) looks like. Even more sophisticated 3 dimensional fractals are used by such film makers as George Lucas of Star Wars fame for special effects within their movies. The genesis sequence in "Start Trek, Wrath of Khan" was done using them.

Finally, I'll still be using SCREEN 1 or 2 for the example programs in this chapter—if you have better graphics hardware, you now know how to use it!



9-1 The Koch snowflake.



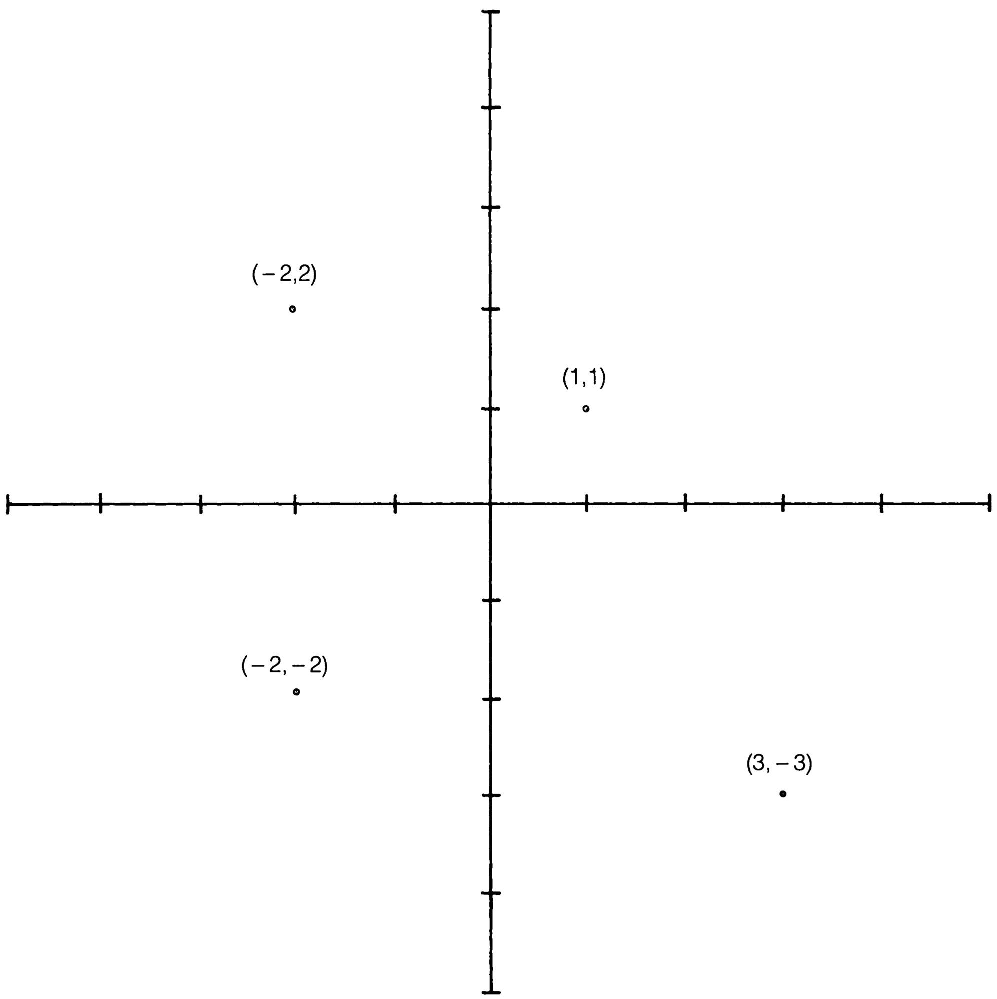
9-2 A fractal coastline.

The WINDOW command and some simple math

The screen is normally numbered, with (0,0) being the top left hand corner. This is obviously inconvenient for doing mathematics. Mathematics uses an X-Y system, with X measuring how much across you are from a central point, the origin, and Y measuring how much up or down from the center you are. For example, Fig. 9-3 plots a few points on the plane.

The command WINDOW sets up new coordinates that you can use in any of the graphics commands that mimics the coordinates in the plane. For example:

```
WINDOW (-320,100) -(320,-100)
```



9-3 Cartesian plane.

sets up a new coordinate system with the coordinates of the top left hand corner being (−320,100), and the bottom right corner being (320,−100). After this command:

```
PSET (-320,100)
PSET (320,100)
PSET (320,-100)
PSET (-320,-100)
PSET (0,0)
```

would illuminate the four corners in a clockwise order, starting from the top left, and then it illuminates the center of the screen. This is because whenever you issue a WINDOW command followed by a graphics command, QB automatically finds the pixel that corresponds to your coordinates, rounding if necessary.

In general, the WINDOW command looks like this:

```
WINDOW (LeftX,TopY) -(RightX,BottomY)
```

where LeftX is a single-precision real number, that will represent the smallest X coordinate (left most),TopY a single precision number for the largest Y (top), etc. WINDOW (−1E38,1E38) −(1E38,−1E38) gives you the largest possible scale, which means the smallest amount of detail. Large X and Y changes are needed to light up adjacent pixels! On the other hand:

```
WINDOW (-319,99) - (320,-100)
```

not (320,99), gives you a perfect match.

The next program sets up an X-Y axis in SCREEN 2 that allows numbers on the axes satisfying the equation:

$$-5 <= X,Y <= 5:$$

```
SCREEN 2
WINDOW (-5,5) - (5,-5)

LINE (-5,0) - (5,0)                    'X Axis
LINE (0,5) - (0,-5)                 'Y Axis

' Now to label the axes add:

LOCATE 12,1: PRINT "X -Axis";
FOR I = 1 TO 6
 LOCATE I,42
 PRINT MID$("Y Axis",I ,1);
NEXT I
```

I LOCATE on the 12th text line and 42nd column in order to be a bit away from the center. Finally, you might want to add little lines as tick marks:

```
FOR I = -5 TO 5
 IF I <> 0  THEN
   LINE (I,.2) - (I, -.2)
   LINE (-.1,I) - (.1,I)
 END IF
NEXT I
```

This fragment is actually the most interesting. It takes into account that, with this WINDOW setting, a horizontal line as long as a vertical one needs roughly twice the vertical distance. In terms of pixels, in this WINDOW, a line from −.2 TO .2 takes up $^{.4}/_{10}$ ($= ^{1}/_{25}$) of the rows (= 8 pixels = one text line). A line from −.1 to .1 takes up $^{.2}/_{10}$ ($^{1}/_{50}$) 12 pixels, or a little less than 2 text columns.

Labeling objects after a window command is made easier with two commands: PMAP and POINT. After a WINDOW command, PMAP(X,0) gives the ordinary pixel column coordinate for the point X, and PMAP(Y,1) gives you the ordinary row coordinate corresponding to Y. As you can imagine, these commands make captioning graphics easier. If you want to mark a specific place on the screen after a WINDOW command: go to the Immediate window, issue two PMAP commands to find its ordinary coordinates, and then convert these to text rows and columns by dividing by the appropriate number (8 for rows, 8 or 16 for columns). Finally, use LOCATE to put the text where you want it.

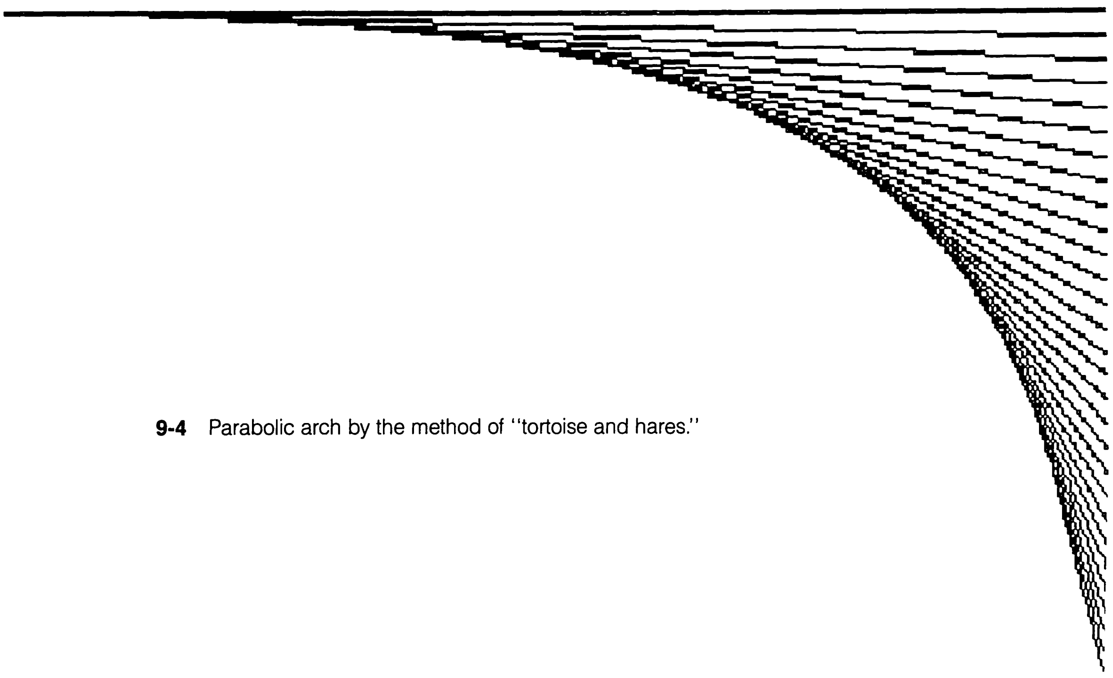
Similarly, you can do it in the reverse order: PMAP(Column,3) gives you the column (X) coordinate in your WINDOW, whereas PMAP(RowCord,4) gives you the Y coordinate in the WINDOW of points given in their ordinary coordinates.

POINT works with the last point referenced (LPR). POINT(0) gives you ordinary column coordinate of the LPR, POINT(1) the ordinary row coordinate. On the other hand, after a WINDOW command, POINT(2) gives you the X coordinate and POINT(3) the Y-coordinate, using the window's X-Y coordinates.

The WINDOW command makes graphing any mathematical function easy. The only problems come in deciding the maximum and minimum values to use for the WINDOW statement, which often takes calculus. However, as before, QB will clip any figure that is off the axis, so no problems result with setting the wrong scale.

Here is a program for a cosine graph:

```
DEFSNG A-Z
TwoPie! = 8*ATN(1)
SCREEN 2
WINDOW (-TwoPie,1) - (TwoPie,-1)

For I = -TwoPie! TO TwoPie! STEP .01
  PSET (I,COS(I))
NEXT I

END
```


9-4 Parabolic arch by the method of "tortoise and hares."

I'll leave it to you to put in the axes and mark them. I'll also leave it to you to experiment with other functions—ones that you may need for school or work.

Now that you know about the WINDOW command, you can now get to some serious picture drawing. The first method I want to show you depends on the following simple idea: imagine two points in the plane, say with a tortoise at one and a hare at the other, chasing each other. As the second point, the hare, moves, draw the line connecting the first points, the tortoise's old position to the new position of the second point. Now move the tortoise down this line a little bit, say 10% of the way. Continue the process. Here's what you get if the second point (the hare) moves directly down 5 units, and the first always moves 10% of the way (Fig. 9-4).

To implement this idea you need a formula that calculates the new coordinate. Suppose first that you want the point 1/2 way down on the line connecting say (50,100) to (100,200). It's pretty obvious that it has to be (75,150). Suppose, though, you wanted a point that was only 10% of the way along this line. This turns out to be (55,110), since, in some sense, you have 50 X units to move, and 100 Y units to move. In general, the formula to move on a line connecting (x1,y1) to (x2,y2) is:

```
(1 - t) *x1 + t*x2
```

for the new x coordinate, and:

```
(1 - t) *y1 + t*y2
```

for the new y coordinate. Here t is the % moved, expressed as a decimal. I like to think of this as a weighing formula.

Given this, here is the program that draws the parabolic arch given on page 246:

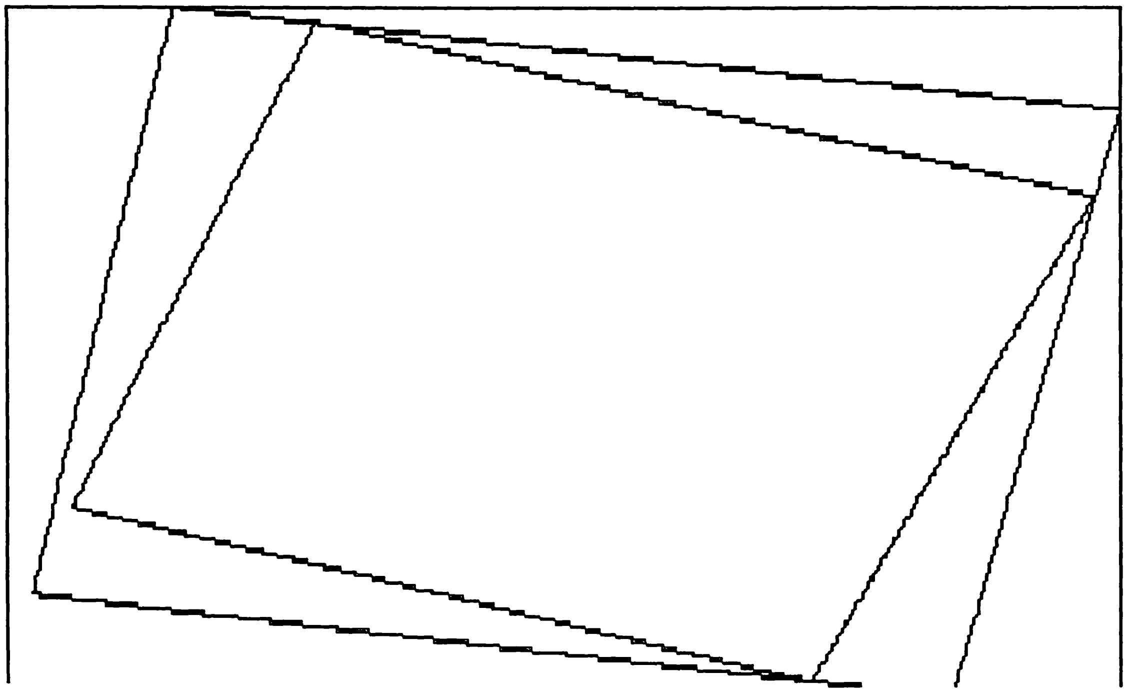
```
'CH9\P1.BAS
' a parabolic arch

DEFSNG A-Z
SCREEN 2
WINDOW (-300, 100)-(300, -100)

x1 = -300: x2 = 300
y1 = 100: y2 = 100

PerCent = .1

DO UNTIL y2 < -100

  y2 = y2 - 5                                 'down five units
  LINE (x1, y1)-(x2, y2)                      ' connect the points

  x1 = (1 - PerCent) * x1 + (PerCent * x2)    'move down the line
  y1 = (1 - PerCent) * y1 + (PerCent * y2)

LOOP

END
```

Although not a bad start, you don't really start getting results until you add more animals, or points. Imagine the four animals start at the corners of a square. The first animal chases the second, the second chases the third, the third the fourth, and the fourth chases the first. Figure 9-5 shows you what you get after only two moves. Obviously what is happening is that each square is both rotating and shrinking. If you continue this process, then you get the first figure given in the last chapter. Before I give you the program, though, I need to remind you of one more formula: the so called distance formula. This says that the distance between two points (x1,y1) and (x2,y2) in the plane is:

$$\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

or in QB you could use the following fragment:

```
FUNCTION Dist(x1,y1,x2,y2)

A = (x2-x1)*(x2-x1)
B = (y2-y1)*(y2-y1)

Dist = SQR(A+B)

END FUNCTION
```


9-5 Rotating squares.

The purpose of having a distance function is to know when to stop—to know when the animals are close enough.

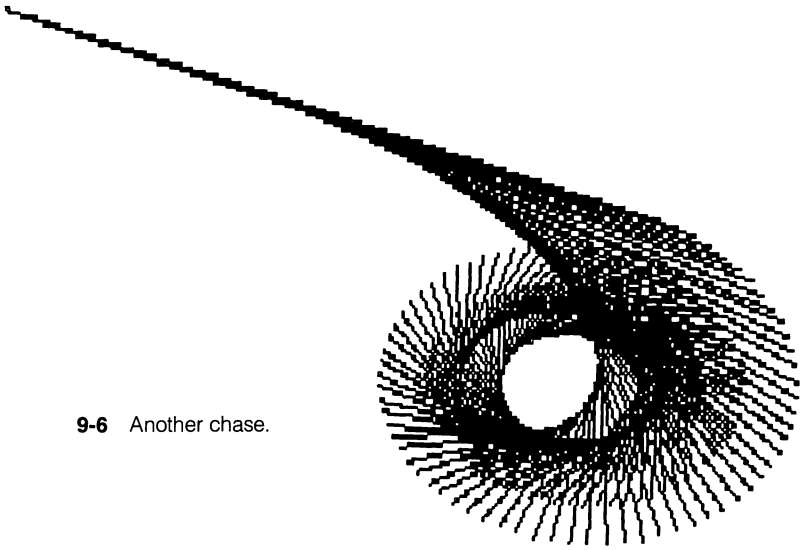
This is the program that drew Fig. 8-1—it needs the DIST function given on the preceeding page:

```
DECLARE SUB move (a!, B!, T!)
DECLARE FUNCTION DIST! (x1!, y1!, x2!, y2!)

' C98\P2.BAS
' Nested (rotated) squares

DEFSNG A-Z
SCREEN 2
WINDOW (-320, 200)-(320, -200)

T = .05                    'Percentage moved if 5%
x1 = -320: y1 = 200
x2 = 320: y2 = 200
x3 = 320: y3 = -200
x4 = -320: y4 = -200

DO UNTIL DIST(x1, y1, x2, y2) < 10              'A
```

```
   LINE (x1, y1)-(x2, y2)                          'B
   LINE -(x3, y3)
   LINE -(x4, y4)
   LINE -(x1, y1)

   CALL move(x1, x2, T)                            'C
   CALL move(y1, y2, T)
   CALL move(x2, x3, T)
   CALL move(y2, y3, T)
   CALL move(x3, x4, T)
   CALL move(y3, y4, T)
   CALL move(x4, x1, T)
   CALL move(y4, y1, T)

LOOP

SLEEP

END

FUNCTION DIST (x1, y1, x2, y2)

a = (x2 - x1) * (x2 - x1)
B = (y2 - y1) * (y2 - y1)

DIST = SQR(a + B)

END FUNCTION

SUB move (a, B, T)
a = (1 - T) * a + T * B
END SUB
```

A) This stops the process when the points get close enough.
B) I can't use the box command, because the square is rotated.
C) This block finds the new coordinates for each of the four points. By adding more parameters, I could have made the Move sub-program make the changes one point at a time, instead of one coordinate at a time.
D) This is the move function discussed earlier.

If you imagine the animals are moving independently along curves then the kind of pictures can be even more dramatic. Figure 9-6 is one of the simplest ones. In this picture you should imagine one point is constantly moving around a circle around the origin, while the other point chases it by moving along a line of sight. To write a program to do this, or to construct one whose results are even more dramatic (chases along more complicated curves), you'll need formulas for the curves—that's the subject of the next section. After reading that section you'll have no trouble writing a program to draw the figure.


9-6 Another chase.

Polar coordinates

Most complicated mathematical curves are more easily described using polar coordinates. With polar coordinates one describes the position of a point by saying how far it is from the origin, and what angle a line connecting the origin to it makes with the positive x-axis. Figure 9-7 shows this.

To go from polar coordinates to X-Y coordinates, one uses the formula:

$$X = R*COS(Angle)$$

and:

$$Y = R*SIN(Angle)$$

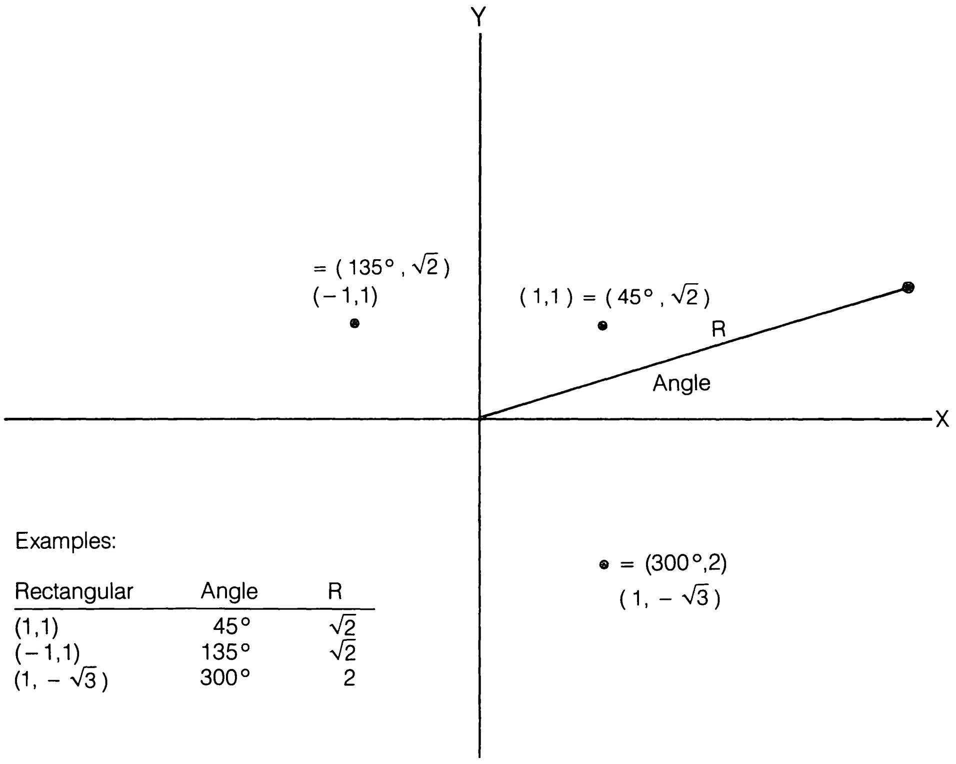
where Angle is the angle indicated in the figure above. These come from dropping a perpendicular to the X-Axis and making a right triangle.

To go from X-Y coordinates to polar coordinates, use:

$$R = \sqrt{X^2 + y^2}$$

and the angle is ATN(Y/X), unless X is zero.

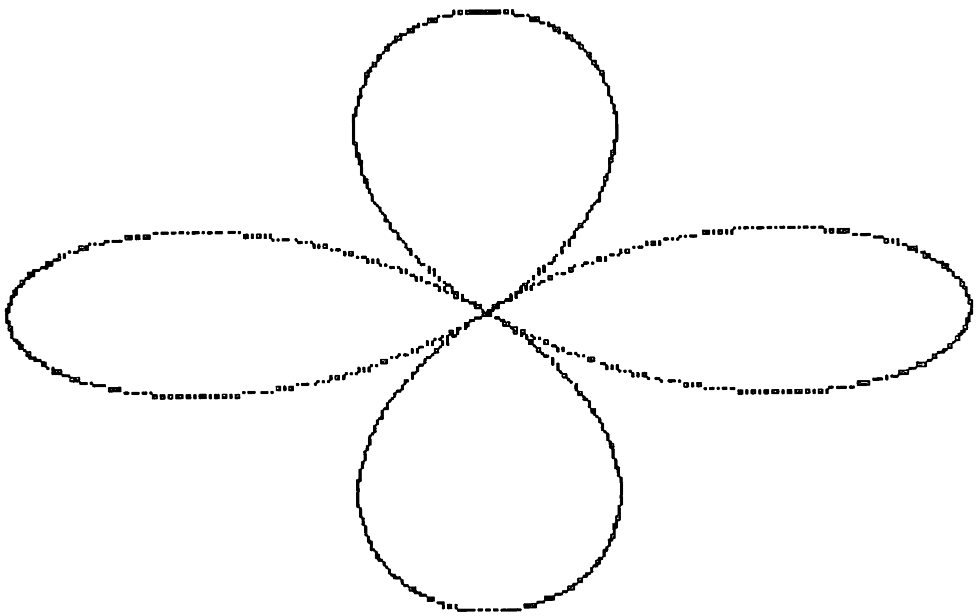
The point of polar coordinates for you is that the equation of a curve may have a much simpler formula than in rectangular (X-Y) coordinates. For example, the equation of a circle of radius 10 around the origin is simply $R=10$ (instead of $X^2 + Y^2 = 100$). As a more interesting example you can easily draw objects like a four leaf clover. Figure 9-8 shows this. The formula for this is R = COS-(2*Angle) as the angle runs from 0 to 2π ($0°$ to $360°$). I won't bother giving the



Examples:

Rectangular	Angle	R
(1,1)	45°	$\sqrt{2}$
(−1,1)	135°	$\sqrt{2}$
(1, − $\sqrt{3}$)	300°	2

9-7 Polar coordinates.



9-8 A four leaf clover.

rectangular X-Y version. Combine this formula with the conversion formulas for X and Y given above, and you have the following simple program that draws a four leaf clover:

```
'CH9\P3.BAS
DEFSNG A-Z

TwoPie = 8 * ATN(1)
SCREEN 2
WINDOW (-2, 2)-(2, -2)

FOR I = 0 TO TwoPie STEP .01

 R = COS(2 * I)
 X = R * COS(I)
 Y = R * SIN(I)
 PSET (x, y)

NEXT I

END
```

Don't be surprised if this takes a bit of time. After all, it requires a few thousand sine and cosine computations. For that reason there's a common way to speed up polar graphs: instead of going from 0 to 2π, use the symmetry in the sine and cosine function. You really only have to compute sines and cosines from 0 to π if you add a PSET($-$x,$-$y). This trick works because:

SIN(180+Angle) = $-$SIN(Angle) and COS(180+Angle) = $-$COS(Angle)

In general, of course, you have to know the formula for the object in question. Over the years people have collected this information and it's readily available in libraries.

The following demonstration program collects a large number of examples through the use of a random number generator. It also uses the following observation: if you slightly twist the conversion formulas from, for example:

X = R*COS(Angle)

to:

X = R*COS(RandomInteger*Angle)

then the pictures are even more dramatic.

The following program is long but nothing in it is very difficult. The most important is the following SUB that chooses the function to be graphed:

```
FUNCTION RofTheta (Theta, Curv%)

SHARED FFActr() AS INTEGER                    'A
SELECT CASE Curv%
```

```
 CASE 1                                    'B
   RofTheta = 1 + (2 * COS(2 * Theta))
 CASE 2
   RofTheta = COS(4 * Theta)
 CASE 3
   RofTheta = Theta / 2
 CASE 4
    RofTheta = (2*SIN(FFActr(3)*Theta)) -(2*COS(FFActr(4)*Theta))
 CASE 5
   RofTheta = 3 * COS(FFActr(1) * Theta)
 CASE 6
   RofTheta = 1 + (FFActr(1)*COS(FFActr(2) * FFActr(3) * Theta))
 CASE 7
   RofTheta = 1 + (FFActr(1) * COS(FFActr(2) * Theta))
 CASE 8
   RofTheta = (FFActr(1) * SIN(Theta)) / (2 + COS(Theta))
 CASE ELSE
   A = (FFActr(1) * COS(FFActr(3) * Theta))
   B = (FFActr(2) * SIN(FFActr(4) * Theta))
   RofTheta = A + B
END SELECT

END FUNCTION
```

A) This array stores the various fudge factors—random integers that I'll set up elsewhere.
B) The first three cases set up three un-fudged curves: a loop the loop, a four leaf clover, and a spiral. The other cases are affected by the random integers used as fudge factors.

Now, to graph the curves, we modify the normal polar graphing module to allow twisted coordinates. The parameter Tf% acts as a switch to determine whether to twist the coordinates or not.

```
SUB PolarPlot (Curve%, Tf%)

SHARED Pie, limit%, FFActr() AS INTEGER

  ' This procedure plots the graph using either normal or twisted
  ' polar coordinates

  CLS
  FOR I = 0 TO Pie STEP .005
    IF Tf% = TRUE THEN                        'Fudge it
     X = RofTheta(I, Curve%) * COS(FFActr(5) * I)
     y = RofTheta(I, Curve%) * SIN(FFActr(6) * I)
    ELSE                                           'reg polar
     X = RofTheta(I, Curve%) * COS(I)           'coord
     y = RofTheta(I, Curve%) * SIN(I)
    END IF
   IF X > limit% OR y > limit% THEN STOP
   PSET (X, y): PSET (-X, -y)
 NEXT I

END SUB
```

Here's the rest of the program:

```
DECLARE SUB PolarPlot (Curve%, I%)
DECLARE FUNCTION AskIfComplicated% ()
DECLARE SUB MakeFudgeFactors (A%())
DECLARE FUNCTION MaxSize% (C%)
DECLARE FUNCTION RofTheta! (Theta!, C%)

' A polar coordinate demo

DIM FFActr(1 TO 7) AS INTEGER          'stores 8 random integers

CONST TRUE = -1
CONST FALSE = NOT TRUE

Pie = 4 * ATN(1)

SCREEN 2

PRINT "Press X to eXit any other key to continue.";
YN$ = INPUT$(1)

DO UNTIL YN$ = "X"

  CLS
  Complicated% = AskIfComplicated%

  ' Now CALL a procedure that gets 7 random integers
   CALL MakeFudgeFactors(FFActr())

  ' This block `sizes' the window depending on the chosen curve
   CurveNumber% = FFActr(7)
   limit% = MaxSize%(CurveNumber%)
   WINDOW (-limit%, limit%)-(limit%, -limit%)

CALL PolarPlot(CurveNumber%, Complicated%)
LOCATE 1, 1: PRINT "Press any key to continue - X to eXit.";
YN$ = INPUT$(1)

LOOP
CLS
END

FUNCTION AskIfComplicated%

  CLS
  PRINT "If you want the picture to be really complicated ";
  PRINT "press Y."
  YN$ = INPUT$(1)

  IF UCASE$(YN$) = "Y" THEN
     AskIfComplicated% = TRUE
  ELSE
     AskIfComplicated% = FALSE
  END IF
  CLS
END FUNCTION

SUB MakeFudgeFactors (A%())

' This sub creates an array of 7 random integers between 1 and 9
' local variables X,N%

CLS

INPUT "Enter a positive integer as `seed'"; N%
RANDOMIZE N%

FOR I = 1 TO 7
 A%(I) = 1 + (8 * RND(1))
NEXT I

END SUB

FUNCTION MaxSize% (C%)
' This function determines the sizes of the curve and uses that
'COS and SINe are both at most 1

SHARED FFActr() AS INTEGER

SELECT CASE C%
  CASE 1 TO 5
   MaxSize% = 4
  CASE 6 TO 8
   MaxSize% = 1 + FFActr(1)
  CASE ELSE
   MaxSize% = FFActr(1) + FFActr(2)
END SELECT

END FUNCTION
```

Figure 9-9 is an example of what you can get out of this program.

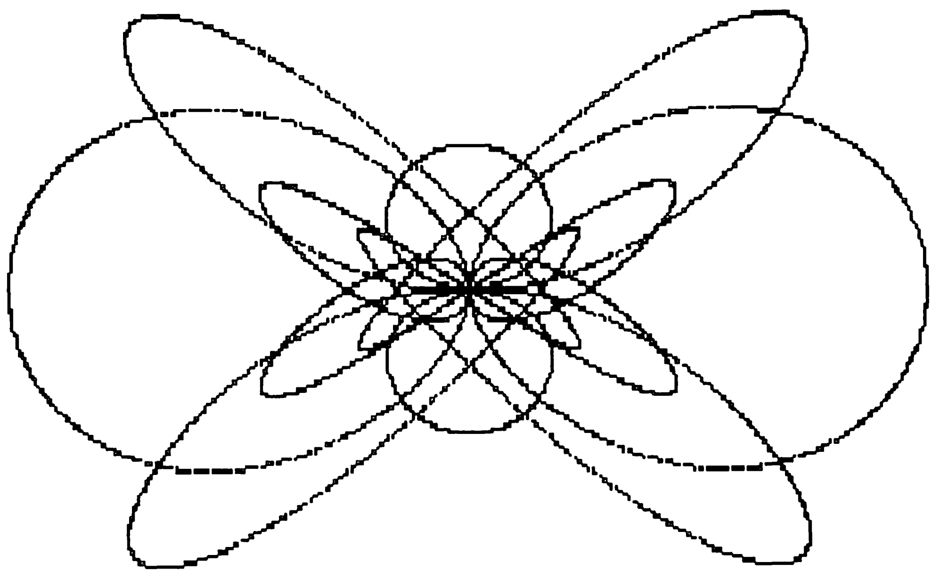
Fractals

Besides some trigonometry this section depends on understanding recursion.

Benoit Mandelbrot of IBM, who coined the term fractal, and is doing much to show how useful the idea is, begins his book *The Fractal Geometry of Nature*, (Freeman 1982) with the following:

Why is geometry often described as "cold" and "dry"? One reasons lies in its inability to describe the shape of a cloud, a mountain, a coastline, or a tree. Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line.

A little bit later he goes on to say that these objects are most often "identical at all scales." This is the simplest way to get at many fractals: objects that, no matter



9-9 A twisted polar graph.

how powerful the magnifying glass, remain essentially the same. The large scale structure is repeated, ad infinitum, in the small. One of Mandelbrot's standard examples is a coastline: from an airplane, from afoot or using a magnifying glass—you get the same pattern on an ever smaller scale.

A pseudo code description of a general fractal is:

```
Draw the object in the large
Replace pieces of the large object by smaller versions of itself.
```

This is obviously a description of a recursive process. Before we get to any of the classic fractals look at Fig. 9-10. As you can see this figure consists of squares the corners of which are replaced by still smaller squares. The pseudo code for this program might be something like:

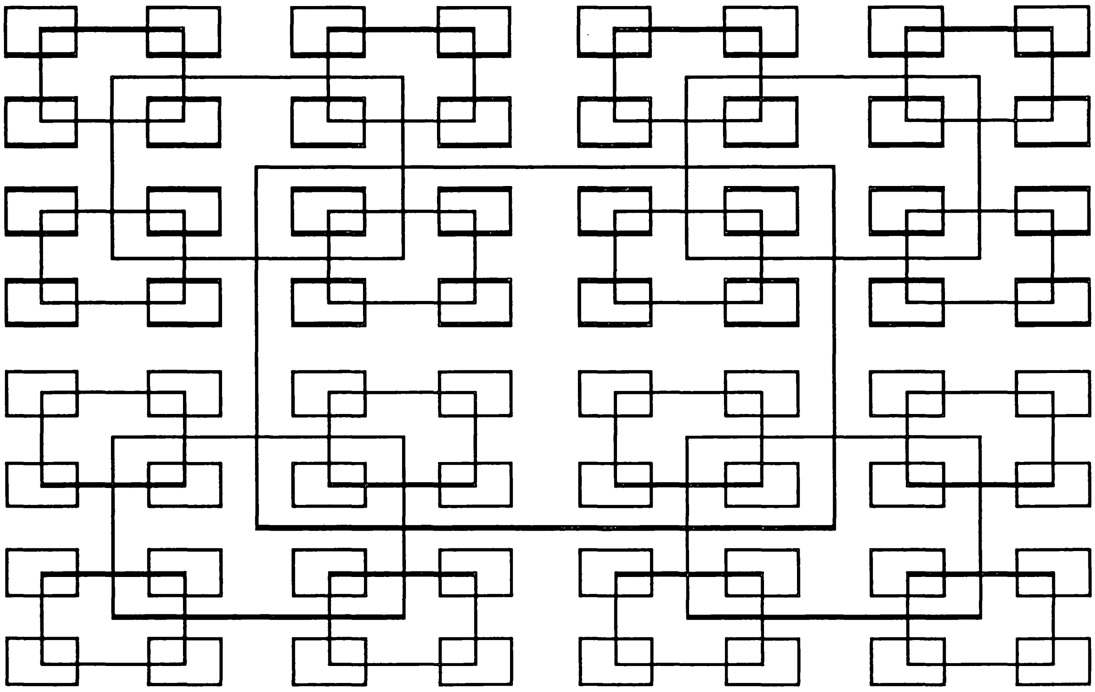
```
SUB Draw A Square
    At each corner of the square
    Draw a square of smaller size,
    CALL Draw A square unless the squares are already too small
END SUB
```

Here is a program that implements this outline:

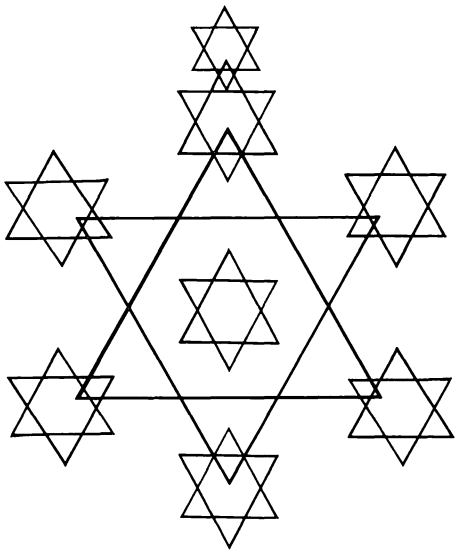
```
'CH9\P5.BAS
' recursive squares

DECLARE SUB square (x%, y%, size%)
DEFINT A-Z
```

```
SCREEN 2
WINDOW (-2000, 2000)-(2000, -2000)

CALL Square(-1000, 1000, 2000)

END

SUB Square (x, y, Size)                                    'A

 IF Size < 50 THEN EXIT SUB

 LINE (x, y)-(x + Size, y - Size), , B                     'B

 CALL Square(x - Size / 4, y + Size / 4, Size / 2)         'C
 CALL Square(x + Size - Size / 4, y + Size / 4, Size / 2)
 CALL Square(x - Size / 4, y - Size + Size / 4, Size / 2)
 CALL Square(x + Size - Size / 4, y - Size + Size / 4, Size / 2)

END SUB
```

A) This gives a way to terminate the recursion.
B) This draws the large square.
C) This is the recursive call. There are four new corners, each one is moved 1/4 of the size of the previous, in or out from the previous one, and is 1/2 as big.



9-10 A recursive shrinking square (incomplete).



9-11 The start of a Koch snowflake.

Figure 9-11 is a screen dump of the start of one of the first fractals to be discovered. It's the beginning of a fractal called the Koch snowflake that you saw in the first figure in this chapter. As you can see, Fig. 9-11 consists of a Star of David repeated on an ever smaller scale. The key to programming this is to notice that if you start from the center of the Star of David of size, say, eight, then each vertex has either a one, two, or four unit shift in the X or Y level.

Once you understand this, then, to write the program only requires setting up an array that, on each call, holds the current values for the 12 vertices. Here is the program:

```
'CH9\P6.BAS
' The Koch snowflake

DECLARE SUB Koch (x%, y%, size%)

CLEAR , , 10000

DEFINT A-Z
SCREEN 1
COLOR 0, 2
WINDOW (-100, 100)-(100, -100)
CALL Koch(0, 0, 120)

END

SUB Koch (x, y, size)

 'local variables are X(),Y(),shift,i,colnum
```

```
DIM x(12), y(12)

 IF Size < 5 THEN EXIT SUB
 shift = size / 8
 x(1) = x: y(1) = y + 4 * shift
 x(2) = x + shift: y(2) = y + 2 * shift
 x(3) = x + 3 * shift: y(3) = y(2)
 x(4) = x + 2 * shift: y(4) = y
 x(5) = x(3): y(5) = y - 2 * shift
 x(6) = x(2): y(6) = y(5)
 x(7) = x: y(7) = y - 4 * shift
 x(8) = x - shift: y(8) = y(5)
 x(9) = x - 3 * shift: y(9) = y(5)
 x(10) = x - 2 * shift: y(10) = y
 x(11) = x(9): y(11) = y(2)
 x(12) = x(8): y(12) = y(2)

 LINE (x(1), y(1))-(x(5), y(5))                    'A
 LINE -(x(9), y(9))
 LINE -(x(1), y(1))

 LINE (x(3), y(3))-(x(7), y(7))                    'B
 LINE -(x(11), y(11))
 LINE -(x(3), y(3))

 CALL Koch(x, y, 2 * shift)                   'C
   FOR i = 1 TO 12
     CALL Koch(x(i), y(i), 3 * shift)              'D
   NEXT i

END SUB
```

A & B) This draws the star.
C) This gives you ever smaller stars in the middle of each star.
D) This gives you a star at each of the twelve vertices.

To understand the remaining fractal curves, observe that there's another way to think of the Koch snowflake. What you're doing is replacing each straight line segment by a line segment that looks like Fig. 9-12.

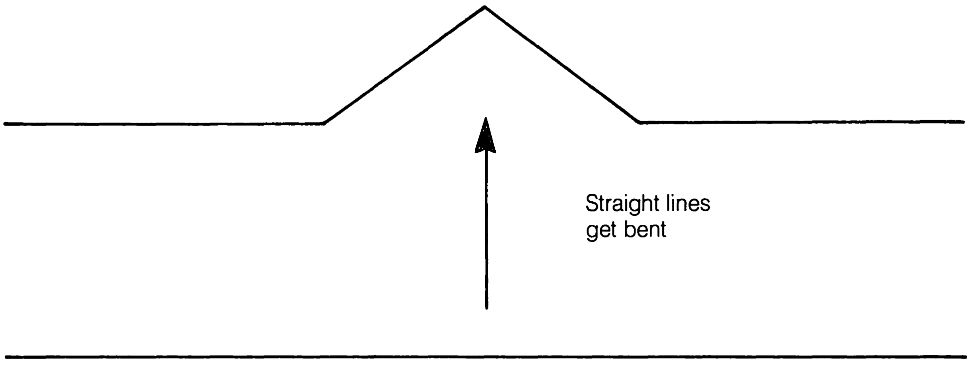
This idea of continually replacing a straight line by a bended line is the key to the next two curves. In the first, usually called a C-curve, we replace each straight line by a bend as in Fig. 9-13.

This eventually gives you a figure that looks like Fig. 9-14.

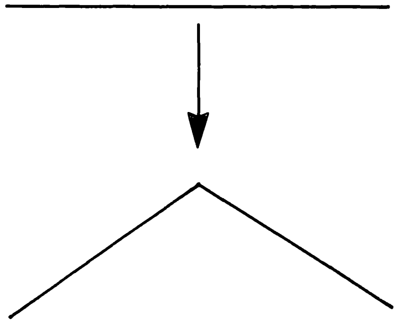
For the next fractal, called the Dragon curve, you modify the C-Curve by putting the bends on opposite sides. The replacement parts alternately go out and in. Figure 9-15 is a picture of what you get.

The pseudo code for both these programs is the same:

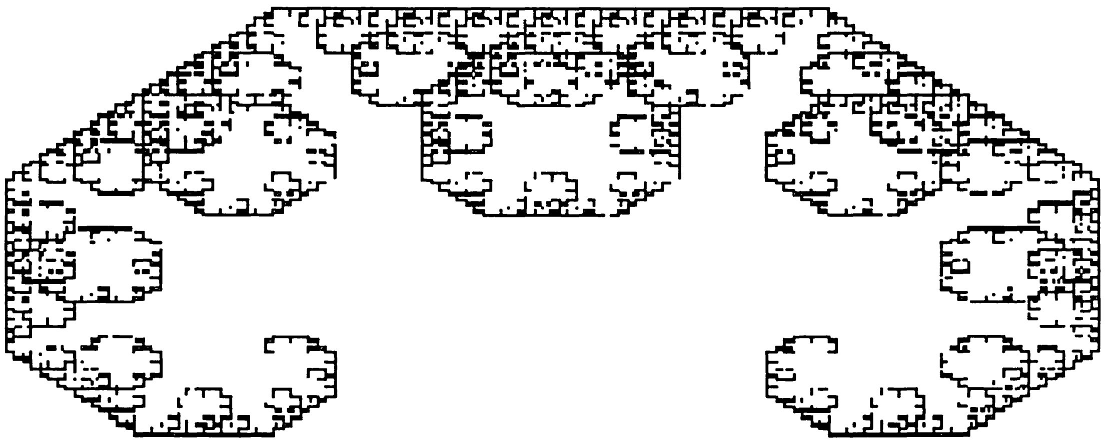
```
SUB DrawAFractal with MakeABend
If the line isn't too small
```



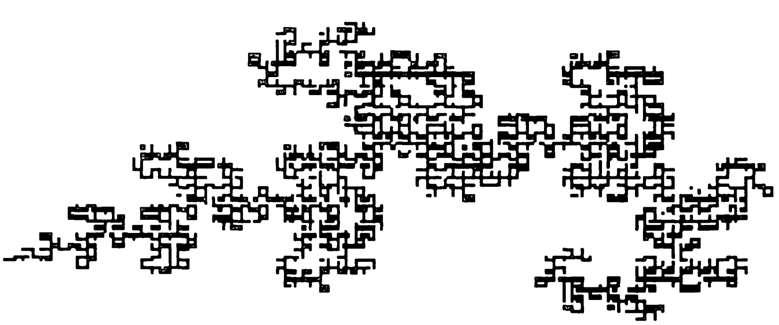
9-12 The twist for the Koch curve.

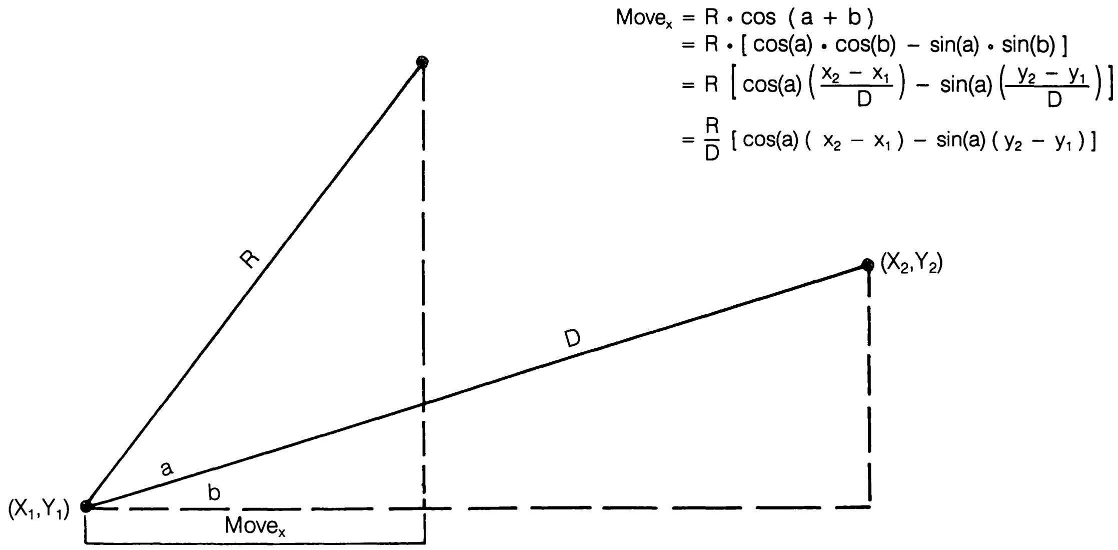


9-13 Building block for the C-curve.



9-14 C-curve.


9-15 The dragon curve.


$\text{Move}_x = R \cdot \cos(a + b)$
$= R \cdot [\cos(a) \cdot \cos(b) - \sin(a) \cdot \sin(b)]$
$= R \left[\cos(a)\left(\frac{x_2 - x_1}{D}\right) - \sin(a)\left(\frac{y_2 - y_1}{D}\right)\right]$
$= \frac{R}{D}[\cos(a)(x_2 - x_1) - \sin(a)(y_2 - y_1)]$
R
D
a
b
(X_1,Y_1)
(X_2,Y_2)
Move_x
9-16 The trigonometry of finding a new point.

Replace the line by the bended one.
CALL SUB for the smaller line

The only point remaining is to describe, mathematically, making a bend. This is where trigonometry comes in. What you need is a formula that, given a line connecting any two points, and angle, finds the coordinate of the new point. Look at Fig. 9-16.

Notice that if the angle is 45°, as it is in the C and Dragon curves, then the size of the spike is:

COS(45°)*D = (SQR(2)/2)*D

because the triangle is a 45,45, 90 right triangle. Using this, the keys to a program for the C-Curve are the following MoveX and MoveY functions:

```
FUNCTION MoveX! (X1!, y1!, x2!, Y2!)

' local variables: Angle(in radians),D,XShift,YShift

Angle = Radians!(45)

D = Dist(X1, x2, y1, Y2)
R = (SQR(2) / 2) * D

XShift = COS(Angle) * (x2 - X1)
YShift = SIN(Angle) * (Y2 - y1)

MoveX = R / D * (XShift - YShift)

END FUNCTION

FUNCTION MoveY! (X1!, y1!, x2!, Y2!)

' local variables: Angle(in radians),D,XShift,YShift

Angle = Radians(45)

D = Dist(X1, x2, y1, Y2)
R = (SQR(2) / 2) * D

XShift = SIN(Angle) * (x2 - X1)
YShift = COS(Angle) * (Y2 - y1)

MoveY! = R / D * (XShift + YShift)

END FUNCTION

' Next we have the main (recursive) SUB:

SUB Curve (X1, y1, x2, Y2)

DEFSNG A-Z
DistPt = Dist(X1, y1, x2, Y2)

IF DistPt < 10 THEN
   EXIT SUB
END IF

nx1 = X1 + MoveX(X1, y1, x2, Y2)        'find the coord for the
ny1 = y1 + MoveY(X1, y1, x2, Y2)        'spike

  LINE (X1, y1)-(x2, Y2), 0             'erase the previous line
  LINE (X1, y1)-(nx1, ny1)              ' make the spike
  LINE -(x2, Y2)

  CALL Curve(X1, y1, nx1, ny1)          'now recurse on the
CALL Curve(nx1, ny1, x2, Y2)          'spikes
```

```
END SUB

'Here's the rest of the program for the `C-curve'.

'A C-Curve

DECLARE FUNCTION Radians! (X!)
DECLARE FUNCTION MoveY! (X1!, y1!, x2!, Y2!)
DECLARE FUNCTION MoveX! (X1!, y1!, x2!, Y2!)
DECLARE FUNCTION Dist! (X1!, y1!, x2!, Y2!)
DECLARE SUB Curve (X1!, y1!, x2!, Y2!)

DEFSNG A-Z
CLEAR , , 12000                          'more stack space
SCREEN 2

CONST PI = 3.14159

WINDOW (-500!, 500!)-(500!, -500!)

X1 = -250: y1 = -200
x2 = 250: Y2 = -200

CALL Curve(X1, y1, x2, Y2)

SLEEP
END

FUNCTION Dist (X1, y1, x2, Y2)
' finds the distance between points
' local variables are: x,y

X = (x2 - X1) * (x2 - X1)
y = (Y2 - y1) * (Y2 - y1)
Dist = SQR(X + y)

END FUNCTION

FUNCTION Radians! (X!)
'converts degrees to radians
Radians! = X! * PI / 180
END FUNCTION
```

The nice thing about these kinds of fractals, is that all you ever have to do is change the MoveX and MoveY function. For the dragon curve, you need to modify them so that they alternate sides each time—a perfect situation for STATIC variables. For example, the next program is the modification of the moveX! function—the moveY! function is similar:

```
FUNCTION MoveX! (X1!, y1!, x2!, Y2!)
' modified mover for Dragon curve
' local variables: Angle(in radians),D,XShift,YShift

STATIC J
J = J + 1
```

```
Angle = Radians!(45)
IF J MOD 2 = 0 THEN Angle = -Angle        'alternate on each call!

D = Dist(X1, x2, y1, Y2)
R = (SQR(2) / 2) * D

XShift = COS(Angle) * (x2 - X1)
YShift = SIN(Angle) * (Y2 - y1)

MoveX = R / D * (XShift - YShift)

END FUNCTION
```

Finally, it's time to show you the coastline program that drew Fig. 9-2. Again, the only difference between this and the two preceding curves is that this time the angle is to be random, as is the size of each spike. To do this, you have to change the Move functions by incorporating a random factor for the angle, instead of 45°, and for the size of the spike, instead of COS(45°)*D. For example in the next listing, (A) and (B) below do this:

```
FUNCTION MoveX! (x1!, y1!, x2!, y2!)
' local variables: Angle(in radians),D,XShift,YShift

Angle = Radians!(15 + 60 * RND(1))              'A

IF RND(1) > .5 THEN Angle = -Angle

D = Dist(x1, x2, y1, y2)
R = (.15 + (.6 * RND(1))) * D                   'B

XShift = COS(Angle) * (x2 - x1)
YShift = SIN(Angle) * (y2 - y1)

MoveX = (R / D) * (XShift - YShift)

END FUNCTION
```

Finally, instead of one Call that acts recursively on a single line, this time you'll build the shoreline out of three recursive calls to three lines that form a triangle. The next program is a fragment that will do this:

```
WINDOW (-500!, 500!)-(500!, -500!)

x1 = -250: y1 = 200
x2 = 250: y2 = 200

CALL Curve(x1, y1, x2, y2)
CALL Curve(x2, y2, 0, -y2)
CALL Curve(0, -y2, x1, y1)
```

Obviously, I have only sampled some of the richness available in the study of fractals—the last section in chapter 11 gives some suggestions on where to go for more information.

10
CHAPTER
Working with files

This chapter shows you how to handle disks and disk files within QuickBASIC. The first section explains the built-in file handling commands. Also, you'll see the SHELL command that lets you invoke DOS commands such as COPY or FORMAT from within QuickBASIC programs, and the COMMAND$ command to read command line information; more on that later. The next four sections explain sequential files. Next I turn to random access files. Finally, I end the chapter with a short discussion of cryptography—how to make the information in files secure from a casual probe.

Interacting with DOS: SHELL and COMMAND$

QuickBASIC has six commands that mimic customary DOS commands:

MKDIR	make a directory
CHDIR	change directory
RMDIR	remove an empty directory
FILES	list files in the current directory
KILL	delete a file
NAME file1 AS file2	renames file1 as file2

You use these commands by following them with a string or string variable. For example: MKDIR "TestDir" would add a sub directory called "TestDir" to the current directory. MKDIR "C: \ "testDir" adds it to the root of the C: drive. The three commands that handle files also accept the normal DOS wildcards. For example, FILES "*.BAS" would give a listing of the files in the current directory with a .BAS extension and KILL "*.*" would delete all the files in the current directory; not to be done casually! NAME can actually do a bit more than the DOS "REN" command: it can copy from one directory in the current drive to another. To do this give the full pathname:

```
NAME "C:\QB45\Test.BAS" AS "C:\Examples\Test"
```

This moves the Test program from the QB45 directory to C: \ Examples \ test.

Although a few of DOS's utility programs are built into QB, usually with slightly different names such as FILES for a variant on DIR/W, and KILL for DEL, most are not. The SHELL command lets you run any .COM, .EXE, or .BAT file from within a QuickBASIC program. For example, if you wanted to display a directory in the form most people are accustomed to see it, use:

```
SHELL "DIR"
```

or

```
SHELL "DIR/P"
```

The general form is:

```
SHELL string
```

where the string contains the name of the standalone program or batch file that you want to execute.

The SHELL command has to be used with care—especially while you're developing the program from within QB. You must have enough memory to keep QB, the current program, a new copy of DOS, and the program you're SHELLing to, all simultaneously in memory. This is less of a problem when you have compiled the program to stand alone—you've freed up the 300 or so K that QB uses. In addition, you must have set DOS's PATH command, or have a disk containing the program that you are SHELLing to, in the current drive. Still the ability to say: SHELL "FORMAT A:" in a crisis can't be overestimated.

If you know DOS well, then the SHELL command, when combined with the pipes and filters briefly discussed in chapter 1, gives you an extraordinarily powerful addition to QB.

Most professional programs allow or require the user to type in additional information when he or she invokes the program. This extra information is usually called command line information. For example, when you say:

```
COPY A:*.* B:
```

the command line information is A:*.* B:. The utility program COPY uses this information to know what to do. Unlike interpreted BASICs, QB makes it easy to read this information. When you say:

program name info1 info2 info3 . . .

and the program was compiled with QB, then the value of the reserved string variable COMMAND$ is the string containing all the info. Since you have already seen how easy it is to write a program that parses a string, the command line information is always available.

Obviously, you need a way to create sample pieces of command line information while developing the program—otherwise you wouldn't have any test data to debug with. This is done from the Run menu. The choice Modify Command$ opens up a dialog box in which you can place a string that would be used in the COMMAND$ command within the program.

Sequential files

These are the analogues in QB to recording information on a cassette tape. It turns out that the analogy is quite close, and so a particularly useful one to keep in mind. For example, the operations on sequential files that are analogous to easy tasks for a cassette recorder, such as recording an album on a blank tape, will be easy, and those analogous to harder tasks, such as splicing tapes together, or especially making a change within a tape, will be hard.

More precisely a sequential file is best when you know that you will:

1. Rarely make changes within the file.
2. Process the information it contains from start to finish, not needing to constantly jump around.
3. Any information that you intend to add to the file can be safely tucked away at the end.

Here's a table of some common operations on a cassette tape and the analogous operations on a sequential text file called SONG in the currently active directory.

PUT THE MACHINE IN "PLAYBACK" AND PAUSE.	OPEN "SONG" FOR INPUT AS #1
PUT THE MACHINE IN "RECORD" AND PAUSE.	OPEN "SONG" FOR OUTPUT AS #1
TAKE THE TAPE OUT	CLOSE #1

Each time DOS sees the OPEN command, it gets ready to send information into or take information out of the file. In computerese it sets up a channel to communicate with the file. What follows the OPEN command is the name of the file you are working with. It must be surrounded by quotes and, unless it is in the current directory, you have to give its full hierarchical path name. You can also use a string variable if its current value is a legal file name. The rules for file names are the rules that DOS imposes: at most eight characters, followed by an optional period, followed by a three character extension. The characters you can use are:

A-Z 0-9 ( ) { } @ # $ % & ^ ! - _ ' / ~

Lowercase letters are automatically converted to uppercase.

Finally, DOS needs a file identifier. This is a number preceded by the # sign that you will use to identify the file. Although you can't change this number until you CLOSE the file, the next time you need the file you can OPEN it with a different ID number. The number of possible files you can have open at one time is limited by DOS to 8, unless you change your CONFIG.SYS file (see chapter 1).

When QuickBASIC processes an OPEN command, it also reserves a file buffer in the computer's memory. Without a buffer, each piece of information sent to or from the disk would be recorded, or played back, separately. Since mechanical operations, like writing to a disk, are much slower than writing to RAM, this would waste a lot of time. Instead when a file buffer fills up, QuickBASIC tells DOS to activate the appropriate drive, and a whole packet of information is sent in a continuous stream to the disk. The number of BUFFERS can also be changed from your CONFIG.SYS file.

The CLOSE command empties the buffer and tells DOS to update the FAT (file allocation table). For this reason, a sudden power outage when you have a file OPEN almost inevitably leads to lost information—occasionally, even a corrupted disk. The CHKDSK/f command is often necessary when this happens.

The PRINT command usually sends information to the screen. A variant, the PRINT # command is one way to send information to a file. Here is an example of a program that sends one piece of information to a file named SONG.

```
'   CH10/P1.BAS
' Writing to a file

OPEN "SONG" FOR OUTPUT AS #1

PRINT #1, "TESTING, 1 2 3"

CLOSE #1

END
```

After the usual remark statements, the first executable statement tells QuickBASIC that we are going to set up a file named SONG having file identifier #1. Note that if a file already exists with the name SONG (in the current directory) then it is erased by this statement. OPENing a file for OUTPUT starts a file anew—the contents of a previous file with the same name are lost.

Next is the statement that actually sends the information to the file. The comma is necessary, but what follows the comma can be anything that might occur in an ordinary PRINT statement. And what appears in the file is a mirror image of what would have occurred on the screen. For example, the file does not contain quotation marks. More precisely, the file will contain the word TESTING followed by a comma followed by the numeral 1, followed by a space, followed by the numeral 2, followed by a space and followed by the numeral 3 and then, although you may not have thought of it at first, the characters that define a carriage-return line feed combination—a CHR$(13) (carriage return) and a CHR$(10) (line feed).

It is extremely important that you keep in mind that the PRINT # command works exactly like the PRINT command. By now you are well aware of the automatic carriage-return line feed combination that follows an ordinary PRINT statement. If I had changed the line to read:

```
PRINT #1, "TESTING, 1 2 3";
```

then the file would contain two fewer characters. The CHR$(13) and CHR$(10) would no longer be there, because the semi-colon, just like for an ordinary PRINT statement, suppresses the carriage return line feed combination.

Why am I harping on this? Well, because the cardinal rule of file handling is: you must know the exact structure of a file if you want to be able to efficiently reuse the information that it contains.

As a third example: suppose I change the line to read:

```
PRINT #1, "TESTING",1,2,3
```

Now the file contains many spaces (CHR$(32)) that were not there before. To see why this must be true, just recall that a comma in a PRINT statement moves the cursor to the next print zone by inserting spaces. Use a comma in a PRINT# statement and the same spaces are placed in your file.

Finally, the command CLOSE, followed by a file identifier, moves (in computerese, it flushes) whatever is in the appropriate file buffer to the disk. The command CLOSE without a file ID flushes all open buffers and it closes all open files.

Once a file is open, QB has a command LOF( ), Length Of File, to tell you how large a file is. To use this command, place the appropriate file identifier number within the parentheses. To see this command at work, and to confirm what I said about the sizes of the various versions of the SONG file, described above, try the following program:

```
' CH10\P2.BAS
' a file tester
' demonstrates the `mirror` image property of PRINT #
```

```
OPEN "Test1" FOR OUTPUT AS #1
OPEN "Test2" FOR OUTPUT AS #2
OPEN "Test3" FOR OUTPUT AS #3

PRINT #1,"TESTING, 1,2,3"
PRINT #2,"TESTING, 1,2,3";
PRINT #3,"TESTING",1,2,3

PRINT LOF(1)
PRINT LOF(2)
PRINT LOF(3)

CLOSE
END
```

If you run this program, then what you'll see is:

```
16
14
47
```

As you can see, the first file does contain 2 more bytes (characters) than the second, to account for the carriage return/line feed combination. The third contains far more than the 14 characters in the phrase "TESTING, 1 2 3". The extra characters, as you'll soon see, are, indeed, spaces.

Reading back information from a file

To read information back from a file, you must OPEN the file for INPUT using its name (the full path name, if it's not in the currently active directory), and give it a file identifier that is not currently being used within the program. It doesn't have to be the same identifier that it was set up with originally. The easiest way to find an unused file identifier is with the command FREEFILE. The value of FREEFILE is always the next unused file ID number. Therefore you merely have to have the statement:

```
FreeFileNumber = FREEFILE
```

at the appropriate point in your program, followed by:

```
OPEN "filename" FOR INPUT AS #FreeFileNumber.
```

Next, you choose a variant on the INPUT command to retrieve the information.

For example, suppose you wanted to read back the file Test1. This contains the word TESTING followed by a comma, followed by the numbers. It ends with the carriage return line feed combination. To decide on the easiest way to read this information back from this file, pretend for a second that you were going to enter this information into the computer via the keyboard. You could not simply say: INPUT A$, because that would pick up only the word "TESTING"; the INPUT command would read information only up to the first comma. So you would likely use LINE INPUT A$, because, as you know, the LINE INPUT command disregards any spaces or commas that may have been typed—it accepts all the information typed until the Enter key was hit. The carriage return-line feed combination corresponds to the Enter key.

Listing 10.3 is a program to read back and print out the contents of the file named Test1.

```
'CH10\P3.BAS
' Reading back a file

OPEN "Test1"  FOR INPUT AS #1

LINE INPUT #1, A$
PRINT A$

CLOSE #1
END
```

As an alternative you could have used this:

```
'CH10\P4.BAS
OPEN "Test1" FOR INPUT AS #1

  INPUT #1,A$,B$,C$,D$
  PRINT A$;" ";B$;" ";C$;" ";D$

CLOSE #1
END
```

or this:

```
'CH10\P5.BAS
OPEN "Test1" FOR INPUT AS #1

  INPUT #1,A$,B,C,D
  PRINT A$;B;C;D

CLOSE #1
END
```

both of which seems rather clumsier. Of course, the last program has recovered the numbers as numbers rather than as strings of numerals. If you have stored numbers in a file, then this is the method to retrieve them.

If you know how many entries there are in a file, then a FOR-NEXT loop is often the easiest way for reading the information back. For example, suppose you're a teacher with a class of 25 students. You know the currently active disk contains a file called GRADES that stores the information about the class in the following form:

Student's Names (first last)	}	
First Exam Grades	}	
Second Exam Grades	}	25 times
Third Exam Grades	}	
Fourth Exam Grades	}	

One useful piece of terminology that will reoccur is that of *fields* and *records*. Think of this file as being made up of twenty-five records, and each record consists of five fields. Usually a program that manipulates this file will read back the information by records, that is, five fields at a time. Each field can be picked up by a single INPUT #, rather than needing the LINE INPUT command. The similarity with user-defined records (chapter 11) is not a coincidence. You'll often find yourself filling in the components of a record from a file.

Knowing the exact format of this file means that you can easily write a procedure that will retrieve this information. First set up a record TYPE:

```
TYPE StudentRecord
    NAME AS STRING * 50
    FirstExam AS INTEGER
    SecondExam AS INTEGER
    ThirdExam AS INTEGER
    FourthExam AS INTEGER
END TYPE
```

Now make up an array of 25 StudentRecords:

```
DIM Grades (1 TO 25) AS StudentRecord
```

and then use this:

```
SUB RetrieveGrade

SHARED Grades()
FileNum = FREEFILE
OPEN "Grades" FOR INPUT AS #FileNum
DIM I AS INTEGER

FOR I = 1 TO 25
 INPUT #FileNum, Grade(I).NAME
 INPUT #FileNum, Grade(I).FirstExam
 INPUT #FileNum, Grade(I).SecondExam
 INPUT #FileNum, Grade(I).ThirdExam
 INPUT #FileNum, Grade(I).FourthExam
NEXT I

CLOSE FileNum
END SUB
```

Now each row of the array Grades contains a record with the name and grades of a student. You could easily incorporate this type of SUB into a program that analyzes the grades.

This file had a simple structure because the cardinal rule remains: You can't do anything with a file, until you bring the information you need from it into memory.

The more complicated the structure of the file, the harder this will be to accomplish. If you can keep the structure of your files simple, then filling up an array is often the method of choice. The reason is that once the information contained in the file is stored in an array, massaging it is easy, usually requiring only a few FOR-NEXT loops to run through it.

For example, suppose I want to write a procedure that can return the average and number absent on each exam. As always, when dealing with an array of records one could have chosen to store this information in two parallel arrays or one string array. In this case if I had chosen to store this information in an array of strings, rather than as in the array of records given above, then I could write a SUB that would take a parameter for the exam number as in this:

```
SUB AnalysisOfExams(ExamNumber)

SHARED Grades$()

NumAbsent = 0
Total = 0

FOR I = 1 TO 25

  IF Grades(I,ExamNumber)="absent" THEN
   NumAbsent = NumAbsent + 1
  ELSE
   Total = Total + VAL(Grades$(I,ExamNumber))
  END IF

NEXT I

PRINT "The number absent was";NumAbsent
PRINT "The class average was";Total\(25 - NumAbsent)

END SUB
```

This procedure is straightforward: the parameter tells the procedure what exam number (column of the array) to look at, and the FOR-NEXT loop runs through each row. The VAL command is needed because the exam grades are stored in a string array.

Adding to an existing file

The GRADE file contained the student's name followed by a list of his or her grades. This is a bit unnatural. A different, more natural, kind of file structure would occur if the teacher enters everything in steps: first the students' names, then, after a while, the results of the first exam, and so on. To write a program to do this, you need a command that lets you add information to the end of an already existing file. The command is: APPEND.

The statement OPEN file name FOR APPEND AS # does three things at once:

1. It OPENs the file; if the file doesn't exist it creates it and sets up the appropriate buffer.
2. It locates the end of the file on the disk.
3. It prepares to OUTPUT to the file at its end.

Recall that if you OPEN an existing file for OUTPUT, then you erase it. Only by using the APPEND command can you add to an existing file.

If I was writing this program for myself, for a single class's records, then I'd probably update this file using a short program that reads the students' names and stores them in an array before the APPENDing. This shows how:

```
'CH10\P8.BAS
DIM StudentNames$(25)
OPEN "Grades" FOR INPUT AS # 1          'A

FOR I = 1 TO 25
 INPUT #1, StudentNames$(I)
NEXT I

CLOSE #1                                'B

OPEN "Grades" FOR APPEND AS #1

 FOR I = 1 TO 25
  PRINT "The grade for ";StudentNames$(I);" is   ";
  INPUT Grade$
  PRINT #1, Grade$
 NEXT I

CLOSE
END
```

A) I'm assuming the file was already created and contains the names of the students.

B) To change a file's status it has to be CLOSEd. Once this command is processed, the next command lets you add to it.

You will probably find yourself writing lots of these quick and dirty programs as you become more familiar with file handling techniques. Although they're never very robust, they do get the job done. You need special techniques to make file handling programs robust—see the next section of this chapter.

Suppose, however, that you were teaching five classes, each with a different number of students. Then I'm sure that the quick and dirty approach is not worthwhile. It's possible to get the classes mixed up, leading to an error message, or even losing a student's grades. To prevent this kind of mishap, write a header to all your files. Use this header to put standard information about the file at the beginning of the file.

To write a usable grade book program, I'd use the first few entries in the file for the name of the class, the semester, the number of exams and the number of students. This kind of information isn't likely to change (see the section on binary file techniques). And it has the added advantage that you can use it to set up the bounds on the loops that will read and process the information contained in the file. The next program is a procedure that I might use to set up a grade book on a disk in the currently active directory:

```
SUB SetUpGrade

' LOCAL variables are: '
FileName$,FileNum,ClassName$,ExamNum,StuNum
CLS

INPUT "What is the name of the file for this class":FileName$
INPUT  "What is the name of the class";ClassName$

FileNum = FREEFILE
OPEN FileName$ FOR OUTPUT AS #FileNum
 PRINT #FileNum,ClassName$
 INPUT "How many exams";ExamNum
 PRINT #FileNum,ExamNum
 INPUT "How many students";StuNum
 PRINT #FileNum,StuNum

CLOSE #FileNum

END SUB
```

Getting serious with sequential files

While FOR-NEXT loops are a convenient way to read back information contained in a file, there are times when they are not practical. There may be too much information in the file, or you don't know what limits to use. You need a way to implement the following outline:

```
While there's information left in the file
    Get next piece of INFO
    Process it
LOOP
```

To do this we need a way to test when we're at the end of a file. The statement in

QB that lets you do this is reasonably mnemonic: it's called EOF( ), End Of File, where the parentheses hold the file ID number. Using this statement, then, a program to read back the information contained in a file set up with PRINT # statements, looks like:

```
FileNum = FREEFILE
OPEN filename FOR INPUT AS #FileNum

    LINE INPUT #FileNum, A$

        DO UNTIL EOF (FileNum)
            ' process line - this would probably be a procedure call or
            ' function call
            LINE INPUT #FileNum, A$
        LOOP
CLOSE #FileNum
```

We use a loop at the top to take into account the unlikely possibility that the file exists but doesn't contain any information, i.e., it is OPENed for OUTPUT, but nothing was actually sent to the file. This fragment is a more or less direct translation of the outline: it picks up a line of data (i.e., all the data up to a carriage return/line feed pair). And it continues doing this until it gets to the end of the file.

This kind of fragment is how you write a simple print formatter for text files, like the README.DOC or PACKING.LST that comes with QB. All you need to know is that you can LINE INPUT each line; that they're not too long or too short.

By the way a lot of people use the DO WHILE form of these loops. They prefer: DO WHILE NOT EOF(1)/LOOP or WHILE NOT EOF(1)/WEND. Since all three forms are equivalent, which you choose is a matter of taste.

One common use of the EOF( ) statement is to read back the information contained in a file character by character. The analogy to keep in mind again is that you read a file as if it were the keyboard. Now, recall that you pick up individual characters from the keyboard with INPUT$( ). It should come as no surprise then that you pick up individual characters from a file with the statement:

```
    INPUT$( , )
```

where the first entry holds the number of characters, and the second holds a file ID. So

```
    SixChar$ = INPUT$(6,#2)
```

picks up 6 characters from a file OPENed for INPUT with file ID #2 and assigns it to a string variable named SixChar$. Just like with the ordinary INPUT$ command, you combine INPUT$(,) with an assignment statement.

Next is a program that reads back the contents of a file character by character, and prints both the ASCII code and the character on the same line:

```
'CH10\P10.BAS
' A `semi' master file reader

INPUT "What is the name of the file";FileName$

FileNum = FREEFILE
OPEN FileName$ FOR INPUT AS #FileNum

A$ = INPUT$(1,#FileNum)

DO UNTIL EOF(FileNum)
 PRINT  A$,ASC(A$)
 A$ = INPUT$(1,#FileNum)
LOOP
CLOSE FileNum
END
```

If you use this program on the files Test1,Test2 and Test3 created in section 2, then you can easily check that the spaces and the carriage return/line feed combinations that I claimed the PRINT # sends to a file are, in fact, there.

Although the INPUT$(,) statement lets you examine the structure of many files character by character, it shouldn't be overused. In fact, it's possible for files created by word processors not to be readable by this method. It turns out that what QB is really doing when it tests for the end of a file with EOF( ) is searching for a Ctrl+Z = CHR$(26). You can see this by trying to use the previous program to read back the file created with the following test program:

```
'CH10\P11.BAS
'demonstrates Cntrl Z (=^Z=CHR$(26) as EOF

FileNum = FREEFILE
OPEN "TEST" FOR OUTPUT AS #FileNum
PRINT #FileNum, CHR$(26)

FOR I = 1 TO 10
 PRINT #FileNum, "The previous program can't ever read this"
NEXT I

CLOSE #FileNum
END
```

Since it can be very important to massage non-ASCII files, such as LOTUS 1-2-3 files, QB has another method of reading back files that, among its other powers, gets around this Ctrl+Z problem. See the section on binary file techniques in this chapter.

In any case, even if a file is readable by this method, it's usually better to think of a file as being made up of fields, possibly grouped into records. Each field is separated from the next by a *delimiter*. That is, a comma or carriage return/line field combination. The delimiter is what lets you use a single INPUT # to pick up the field—and this is much faster than doing it character by character.

The WRITE # and PRINT # USING commands

Since we send information to files as if they were the screen, we again have to solve the problems of:

1. How do you send special characters like quotation marks to the screen.
2. How do you nicely format a file.

As the section heading indicates you solve both these problems with commands analogous to the ones that solved them for the screen. For example, as I briefly mentioned in chapter 4, the WRITE command sends information to the screen surrounded by quotation marks.

```
WRITE "Testing 1, 2, 3"
```

displays:

```
"Testing 1, 2, 3"
```

on the screen. Similarly:

```
WRITE #3, "Testing 1,2, 3"
```

sends exactly this including the quotes and the commas, to the OUTPUT or APPEND file with ID number 3. This is exactly the same as saying:

```
PRINT #3,CHR$ (34);
PRINT #3,"Testing 1, 2, 3";
PRINT #3,CHR$ (34)
```

Note the two semi-colons, to prevent inadvertent carriage return/line feed combinations.

As long as you send individual pieces of information to a file, then the PRINT # and WRITE # command can be used interchangeably.

```
PRINT #FileNum,"Hello"
```

and

```
WRITE #FileNum,"Hello"
```

both put a single piece of information into a file. In either case you can read back the information using the INPUT # command, although the files won't be the same size: the WRITE command adds two quotation marks to the file. It's only when you send more than one piece of information at a time that the differences really emerge. For example, to send three numbers to a file, using:

```
PRINT #FileNum, 1, 2, 3
```

sends a rather large number of superfluous spaces. The command:

```
WRITE #FileNum, 1,2,3
```

sends the appropriate commas to the file—saving space and making it easier to read back the information. It's equivalent to the cumbersome:

```
PRINT #FileNum,1;",";2;",";3
```

Simply put, use WRITE # together with INPUT and PRINT # with LINE INPUT.

The PRINT # USING is rarely used; most people prefer to format the file after the information is read back. However, if you do want to experiment with it, it will, of course, send information to the file, using the same spacing or formatting characters it would send to the screen. To summarize: when you work with a sequential file, output as if it were the screen, and input as if it were the keyboard.

Making a file program robust: Error trapping

Usually when you're testing a program, you don't care if you get a run-time error and your program crashes. However, when an open file is around, then, after a crash, strange things may get written onto your files, or information you need may never be there. Or even if you've thoroughly debugged the program, then someone may try to send information to a full disk, or try to access a file that doesn't exist. To solve these problems you must stop the program when, for example, it faces a full disk. The command that activates error trapping is, as you saw in chapter 5:

```
ON ERROR GOTO . . .
```

where the three dots are for the label or line number that defines the error trap. Now we need to transfer control to a module that:

1. Identifies the problem.
2. If possible, fixes it.

If the error can be corrected then we can use RESUME. However, you can't correct an error if you don't know why it happened. Here's a table giving the error codes that are most common to file handling programs:

25	Device fault. For example, trying to LPRINT when the printer is off.
53	File not found.
55	File already OPENed.
57	Device I/O error. Your hardware is acting up.
61	Disk full. Not enough room to do what you want. One way to prevent this

error is to check the amount of free space available on a disk before you start massive file I/O. See the last chapter.

62 Input past end.

64 Bad File name.

70 Permission denied. The disk you're writing to has the write-protect notch covered.

71 Drive not ready. The door is opened, or where's the floppy?

72 Disk media error. Time to throw out the floppy, or start thinking about the state of your hard disk.

76 Path not found. Asked to OPEN a file on a non-existent path.

You use this information just as it was outlined in chapter 5: somewhere in the program, before the error can occur, place (for example) the statement:

```
ON ERROR GOTO DiskCheck
```

Now write a module like this—labeled by "DiskCheck" or whatever name you chose:

```
Diskcheck:

  ErrorNumber = ERR
  BEEP

  SELECT CASE ErrorNumber

    CASE  53
     PRINT "Your file was not found.  Please check on the "
     PRINT "spelling or call your operator for assistance."

    CASE 61
     PRINT "The disk is full.  Please replace with a slightly"
     PRINT "less used model"  'could SHELL to FORMAT.COM here

    CASE 71
     PRINT "I think the drive door is open - please check"

    CASE 72
     PRINT "Possibly big problems on your hard disk. You'd"
     PRINT "better pray that a low level re-format can help - "
     PRINT "definitely time to call your operator for assistance"

    CASE 57
     PRINT "Possibly big problems on your hardware. You"
     PRINT "definitely should call your operator for assistance"

    CASE ELSE
     PRINT "Please tell the operator (= program author?) that"
     PRINT " error number ";ErrorNumber;"occurred."

 END SELECT
 PRINT "If the error has been corrected press `Y' otherwise "
PRINT "press any other key to END"

Continue$ = INPUT$(1)

IF Continue$ = "Y" or Continue$ = "y" THEN RESUME ELSE END
```

The idea of this error trap is simple—the SELECT CASE statement is ideal. Each case tries to give some indication of where the problem is. And if possible how to correct it. If we reach the CASE ELSE, then the error number has to be reported. In any case, the final block gives one the option of continuing or not.

Error trapping isn't a cure all. Obviously very little can be done about a hard disk crash. On the other hand, if you can SHELL to the FORMAT command (section 1), then not having a formatted floppy around is not a crisis.

As always the PRINT statements in the error trap are likely to mess up the display, unless special precautions are taken as described in the last chapter. A very complete DiskCheck module is your best bet at making a file handling program robust. I usually merge the same module containing this kind of error trap into all my serious file handling programs. Writing a serious file handling program without an error trap is an awful idea.

Some final notes on sequential files: Using devices

The information inside a sequential file is packed tight and hard to change, but that doesn't mean you can't do it. If the changes you're making don't alter the size of the file, then the section on binary files is your best bet. This section will explain some other ways that do not use these techniques.

The APPEND command lets you add information to the end of a sequential file. Suppose now you want to add information to the beginning of a file. First copy the information to a temporary file.

Now one way, the best way in fact, is to use the SHELL command, and use the COPY command to append one file to another. However, suppose you weren't sure the information was to go at the beginning, or that you wanted to remove or replace information already in the file. To do this, imagine what you might do if this modification were to be done to a cassette tape. First, you'd record the words to be added on a separate tape with a little bit of leader; leave some blank tape so you can cut and paste. Then you'd find where on the tape the new information is to go and splice the tapes. For example, to place information at the beginning of an old file you can:

1. Set up a temporary file to hold the new information.

2. APPEND everything in the old file to the temporary file. Now the temporary file contains everything you want. Delete (KILL) the old file, and reNAME the temporary file.

Next, suppose you want to make changes within a file. If the size of the file doesn't change, then binary file techniques are always the best. If it's necessary to change the size of the file, then this is what you have to do: suppose, for example, you want to change all occurrences of the word Turbo Pascal in a file to the word QuickBASIC. In other words, to write your own search and replace function. Then:

1. Read the information in the file into a temporary file, stopping the parade whenever you get to the string "Turbo Pascal."
2. Print the word "QuickBASIC" into the temporary file.
3. Move past the occurrence of the word Turbo Pascal, and continue repeating steps 1 and 2 until EOF.
4. Now KILL the original file, and (re)NAME the temporary file back to the original file's name.

Because you have to read the information back character by character, a program that implements this outline can run for quite a bit of time. The program will run a lot faster, and is actually much simpler, if you know that each occurrence of the string you're searching for was in a separate field. If this were true then you could use a loop that in pseudo-code is:

```
OPEN ORIGINAL FILE
OPEN TEMP FILE
INPUT Field from original file
    DO UNTIL EOF(ORIGINAL)
        IF field <> "Turbo Pascal" THEN
            WRITE IT TO TEMP
        ELSE
            WRITE "QuickBASIC" to TEMP
            INPUT nextfield
    LOOP
KILL ORIGINAL
reNAME TEMP as ORIGINAL
```

The idea of extracting words from a file reoccurs frequently. Suppose you want to index a file. Usually an index throws away the common words and keeps track of the rest. As mentioned in the chapter on recursion, a binary tree is perfect for building an index. Here's an outline of how to write a useful indexing program:

1. Find a list of the most common words. Incorporate this into an ordered array of CommonWords( ).

2. Extract each word—you've seen how to do this in chapter 5.
3. Check to see if it is on the list of common words (binary search).
4. If it is not, add it to the tree.

When the end of the file is reached, the binary tree contains all the information. In fact, by making the nodes of the tree more complicated, you can keep track of where the words occurred and how many times they occurred.

5. Use tree sort from chapter 7 to print out the index.

You can use this idea to write your own programmers cross referencer (XREF). Take the source code of a program and index all but numbers and QB's reserved words.

Here's a list of the devices that QuickBASIC allows you to communicate with:

Name	**Description**	**Input or Output?**
COM1:	First serial port	Both
COM2:	Second serial port	Both
CONS:	The screen	Output only
KYBD:	The keyboard	Input only
LPT1:	First parallel port	Output only
LPT2:	Second parallel port	Output only
LPT3:	Third parallel port	Output only
SCRN:	also the screen	Output only

Although they are not sequential files, QuickBASIC handles communicating with devices as if they were. For example, after:

```
OPEN "LPT2:" FOR OUTPUT AS #3
```

then PRINT#3,A$ sends information to the device, presumably a printer of some type, attached to the second parallel port. If you don't have a second parallel port, or the printer isn't on line, then you'll get an error message. Since LPRINT only sends information to LPT1:, you need this to send information out, if you have printers attached to more than one port on your machine.

The commands that OPEN "COM1:" or OPEN "COM2:" both accept certain parameters controlling baud rate, stop bits, and the like. For more on what these terms mean, consult any book on serial communications. For the full syntax of these commands, consult the on-line help files.

Getting started with random access files

Suppose you were tired of having to search through entire cassettes for certain songs. To avoid this, you decide to put those songs that you want instant access to on individual cassettes. The advantages of doing this are obvious—but there are disadvantages as well. First, to gain more or less instant access to an individual song, you're going to waste a considerable amount of blank tape on each cassette. If to prevent this you decide to create a standard size tape, one that holds, say, four minutes, you're sure to have at least a couple of songs that run more than four minutes.

It's clear that no matter what you do that you'll either waste space, or have a few songs that won't fit. Also, if you single out too many songs for separate tapes, then you increase the number of cassettes you have to store. If you have hundreds of tapes, each containing an individual song, then you're almost back where you started. It can't possibly be easy to find an individual song if you have to search through a hundred tapes. At this point you would probably choose to alphabetize the tapes by some key feature (singer, title . . .), set up an index, or both.

Random access files are stored on a disk in such a way that they have much the same advantages and disadvantages as the song collector's tapes. You gain instant access to individual pieces of information but only at some cost. You must standardize the packets of information involved, so some things may not fit, or space is not efficiently used, and if the file grows too big—with too many pieces of information—then you'll have to set up another file to index the first.

When setting up a sequential file, it's occasionally useful to think of a group of fields as forming a single record. For example, grouping the fields by fives gave a logical and convenient way to read back the information contained in the grade book program. It's worth stressing that this particular grouping was not intrinsic to the file—it's only the way I look at the file. The only intrinsic divisions within a sequential file are those created by the delimiters; commas or carriage return/line feed combinations. When you read back information, you read it field by field—the delimiters acting as barriers. In a random access file, however, the notion of record is built in. A random file is a special disk file arranged by records. This lets you immediately move to the 15th record, without having to pass through the 14 before it. This can and does save a considerable amount of time.

When you first set up a random access file, you specify the maximum length for each record. And when you go to fill up an individual record you may, of course, put in less information than this preset limit—but you may never put in more. So just like the song collector, you might need to prepare for the worst possible situation.

The command that sets up a random access file is analogous to the one for OPENing a sequential file. For example:

```
OPEN "Sample.RND" AS #5 LEN = 100
```

opens a random access file called Sample.RND on the current directory with file ID = 5, and each record can hold 100 characters. Note that unlike the situation for sequential files, you don't have to specify whether you're OPENing the file for INPUT, OUTPUT or APPEND. As you'll soon see, this distinction is taken care of in the commands that manipulate a random access file—an open random access file can be read from and written to simultaneously. You can have any mixture of random access and sequential files OPEN at the same time: the only restrictions are set by how you set up the FILES command in your CONFIG.SYS file. To prevent confusion between file types, I try to use the extension .RND for all my random access files.

Similarly, you close a file opened for random access by the CLOSE command followed by the file ID number. As before, the command CLOSE alone closes all open files. This is especially useful because a sophisticated program for random files often has many files open simultaneously, both sequential and random.

Suppose you wanted to write a random access file that would keep track of your library. You start by designing the form. You decide on five categories:

AUTHOR,TITLE,SUBJECT,PUBLISHER AND MISCELLANEOUS

and after looking over your library you decide on the following limits for the categories:

CATEGORY	**SIZE**
AUTHOR	20
TITLE	30
SUBJECT	15
PUBLISHER	20
MISCELLANEOUS	13

So the total for each record is 98. A random access file to fit this form is set up via (FileNum = FREEFILE as always):

```
OPEN "Mylib.Rnd" AS FileNum LEN = 98
```

Just as each file has an ID number, each record within a random access file has a record number. A single random access file can hold from between 1 to 16,777,216 records! Moreover, you don't have to fill the records in order. As you'll see, you can place information in the 15th record without ever having touched the first 14. The disadvantage of doing this, however, is that QB would automatically set aside enough space for the first 14 records on the disk, even if nothing was in them.

I've been using the word record incessantly—this is no coincidence. One of the main reasons QuickBASIC implemented record TYPEs was to simplify working with random access files. First you set up a type:

```
TYPE BOOKINFO
    AUTHOR AS STRING*20
    TITLE   AS STRING*30
    SUBJECT AS STRING*15
    PUBLISHER AS STRING*20
```

```
        MISCELLANEOUS AS STRING*13
    END TYPE
```

Next, suppose ExampleOfBook was previously DIMmed as being of TYPE BOOKINFO. Then the command:

```
    GET filenum,10,ExampleOfBook
```

would transfer the contents of the 10th record from the random access file into the record variable ExampleOfBook—automatically filling in the correct components of ExampleOfBook.

The command:

```
    PUT filenum,37,ExampleOfBook
```

would send the components of ExampleOfBook to the 37th record of file#2.

This method of sending information to a random access file is unique to QuickBASIC, but it's a very valuable improvement. QB also allows you to use the older, and much clumsier, method that needs what are called field variables. For this and compatibility with interpreted BASIC look at the Programming in BASIC manual or consult a book on interpreted BASIC.

More on random access files: Indices

The record types you create determine the size of the random access file. Since records can hold numbers as well, it's a bit messy to compute the length of a record variable of a given record type. Remember that an integer takes two bytes, a long integer four, etc. QB makes it easy, however, because the LEN command not only gives the length of a string, it also gives you the length of a record. Take any variable of the given type—say you DIM ExampleOfRecord AS ThisType. Next set LenOfRecord=LEN(ExampleOfRecord), and use this to set the LENgth for the OPEN command used to create the random access file.

If you have the information that you want to transfer to a newly created random access file stored in an array of records, then you can use a loop to send the information there. The loop counter determines where to PUT the record. Usually however you set up a variable whose value is the number of the next record you want to read from, or write to.

Similarly, you could read back all the information in a random access file using the EOF flag:

```
    DO UNTIL EOF(filenum)
        I = I + 1
        GET filenum, VariableOfrecordType(I)
    LOOP
```

There are many problems with doing this naively. For one, you're unlikely to want all the information contained in the file at once, and it may not fit anyway. Suppose you have a million records. Also, go back to square one: how do you even know what LENgth to use to OPEN the random access file? While there are many ways to determine this, I prefer to set up another file, that contains this and other vital information about the random access file. At the very least it will contain information about the sizes and types of the fields, possibly names for the fields, and the number of records stored to date. In fact, it may even contain an index of certain keys and the numbers of the records that contain that key.

Indices are vital to a random access file. A data base manager is nothing more than a program to manage random access files. Its speed depends on how the program finds the record containing keyed information. This can only be done effectively through indices. The alternative is to examine the relevant component of each record.

An index can be as simple as a sequential file containing a list of keys, followed by a record number, or a more elaborate ordered one. However, for all but the smallest random access files, indices are best created using binary trees (chapter 9). Recall that we had an array of integers called Arrows% that pointed to the left and right children of a given node. Imagine the contents of the node as the key. Now you're trying to find the record number that contains this key. The solution is simple; add another column to this array Arrows% to hold the record number containing the information keyed by the contents of the node.

In all cases, however, you're likely to read the index into an array or arrays. If you don't like binary trees, then the next best alternative is to quick or Shell sort it on the keys once it gets there. Now you can work with the contents of only a few records at a time. The QB programming manuals work through the construction of an elaborate data base manager, and now is a good time to work through it. When you're done working through it, fix its major flaw by changing its index to a binary tree.

Binary files —some simple file utilities

Binary files are not a new type of file, but a new way of manipulating any kind of file. Binary file techniques let you read or change any byte of a file. Among other features, binary file techniques do not care about any embedded EOF's (Ctrl+Z = CHR$(26)) that the file may have. Recall that it was impossible to read back the file created in section 3 (P6.BAS) using sequential file techniques. The command:

```
OPEN filename FOR BINARY AS filenum
```

sets up a file to be read with these techniques.

Just as with random access files, you can now both read and write to the file. For example, one way to pick up the information from a file open in binary file mode is with the INPUT$(,) command that you saw earlier. It works the same way. The first slot still holds the number of characters and the second the file ID number. For example, the next program gives a module that prints the contents of any file, regardless of any embedded control characters.

```
SUB PrintAFile(A$)
' example of binary input

FileNum = FREEFILE                ' get free file i.d.
OPEN A$ FOR BINARY AS #FileNum

FOR I = 1 TO LOF(FileNum)
 Char$ = INPUT$(1,#FileNum)
 PRINT Char$;
NEXT I
END SUB
```

More often than not, however, you'll want to modify this module by adding some filtering lines. For example, making it strip out the control characters, or those with ASCII codes greater than 127. Once you strip such a file it can, for example, be displayed with the DOS type command, or more easily sent by a modem.

For example, the WordStar word processing program, from which the QB editor gets some of its editing commands, normally stores a file in such a way that when you use the DOS TYPE command you would have trouble reading the file. Similarly, if you had a different word processing program, you might have trouble using it.

Somewhat simplified, here's what WordStar does:

1. It uses certain control codes inside the file, such as Ctrl/B for bold.
2. The end of each line has the high order bit set. This means that if the word processing program had to word wrap a line, the ASCII code of the last character is increased by 128.
3. It uses the carriage return/line feed combination (CHR$(10+CHR$(13)) for hard returns. This means someone has hit the Enter key, rather than the program doing a word wrap.

It's easy to modify the module given above to strip out all formatting control codes, and shift the high order bit back. For those who use WordStar, it will not strip out dot commands—I'll leave the changes needed for that to you. This shows how:

```
SUB StripAFile(A$)
' example of binary input

FileNum = FREEFILE                ' get free file i.d.
```

```
OPEN A$ FOR BINARY AS #FileNum

FOR I = 1 TO LOF(FileNum)

Char$ = INPUT$(1,#FileNum)

SELECT CASE Char$;
  CASE IS < CHR$(32)
   CHAR$ = ""
  CASE IS > CHR$(128)
   CharCode = ASC(Char$)
   CHAR$ = CHR$(CharCode -128)
END SELECT

PRINT Char$;

NEXT I

END SUB
```

Of course, in a more general program you'd probably want to do something more than PRINT the character.

QuickBASIC maintains a file pointer within a file OPENed for BINARY. Each time I use INPUT$, the file pointer moves one position further within the file. The command SEEK is a fast forward and a rewind command combined into one. More precisely:

```
SEEK filenum,position number
```

moves the file pointer for the file with filenum directly to the byte in that position. Any INPUT$( ) would start picking up characters from this location.

SEEK has another use: SEEK(filenum) tells you the position number for the last byte read for either a binary or sequential file. You can also use the SEEK function with random access files. Now it will return the record number of the next record.

To place information within a file OPENed for BINARY, you use a modification of the PUT command. For example:

```
PUT #1,100,A$
```

would place the contents of the string value directly into the file with file ID 1 starting at the 100th byte. The number of characters sent to this file is, of course, given by LEN(A$). The PUT command overwrites whatever was there. If you leave a space for the byte position, but don't specify it in the PUT command:

```
PUT #1, , A$
```

then the information is placed wherever the file pointer is currently located.

The GET command will also work with a binary file. Here though you are best off DIMensioning the variable as a fixed length string. This is because the command:

```
    GET file#,position,A$
```

picks up only as many characters as is the current length of A$. You could use a normal string variable, if you are careful to initialize them to have the correct length.

Now that you know the commands for working on the byte level for a file, you're in a position to write any file utility you may like. You also now can use the information contained in a book like *File Formats for Popular PC Software* by Jeff Walden, (Wiley 1986). Hopefully the manual to your favorite program contains this information as well. Now you can massage the output of any application program. A good example of this may be found in the Dec 13, 1988 issue of *PC Magazine*. This issue had a very useful article on using QB's binary mode to massage LOTUS 1-2-3 files.

Simple ciphers

Now you know that a simple utility program, using binary file techniques, can read back the information contained in any file. So the data contained in your files is readily available to anyone with a compatible computer, a little programming skill, and a copy of your disk. In the next few sections you'll see how to encode a file, so that only people having the right "key" can easily read your file. The methods I show you aren't perfect, but considering how easy they are to implement, they are surprisingly secure. Along the way, you'll see how your computer can help you break some of the most common codes. The computer will do the dirty work, but you'll have to supply the insight.

Actually, cryptographers, unlike in common speech, distinguish between a code and a cipher. To encode for the professional means to change whole words—much the way a company will refer to a product being developed by a code name until it's released. For example, the code name for the IBM PC. Jr. was "peanut." To encipher, on the other hand, means to have rules for changing the individual letters that make up a word. What I'm going to show you here are, to professionals, cipher methods—but like the dictionary I'll occasionally use the terms interchangeably.

First, a little history. All the earliest ciphers that we know about use simple substitutions. For example, Julius Caesar kept his messages secret by taking each letter in the message and replacing it with the one three letters further on. So the letter A would be replaced by D, B by E and so on, until you got to the letters after X. Since X is the 24th letter of the alphabet, you have to wrap around back to the beginning of the alphabet, and X becomes A, Y becomes B, and Z becomes C.

Here's a normal alphabet, and below it a complete Caesar alphabet.

ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC

Actually in Caesar's time the alphabet had fewer letters—23 instead of 26. For example, U and V developed out of V around 1000 years ago and J came around 500 years after that. For example, the sentence "Can you read this," becomes:

Fdq brx uhdg wklv

Shift ciphers go back further than Caesar; one occurs in the Bible. In Jeremiah 25-26, the prophet conceals his prophecy by changing the name of Babylon using a cipher that splits the Hebrew alphabet in half, and replaces the first letter by the middle letter, the second by the middle + 1, and so on.

The next program is a function that shifts any character by any number of characters, wrapping around if necessary:

```
SUB CaesarShift(A$,Shift%)

CharNum = ASC(A$) + Shift

DO UNTIL CharNum >= 0 AND CharNum <= 255              'A
  IF CharNum < 0 THEN
    CharNum = CharNum + 256
 ELSE
    CharNum=CharNum -256
 END IF
LOOP

A$ = CHR$(CharNum)

END SUB
```

A) This little loop wraps around until the shifted character code will give an ASCII character.

It wouldn't be hard to incorporate this procedure into a file encrypter—just pass the procedure the contents of the file character by character. The trouble is that a shift cipher is easy to break—you can even do it by hand. Look at the coded message and run back down the alphabet by steps, shifting the letters back step by step. After at most 25 steps you're done. Here's what you get at each step in the example:

```
Fdq  brx  uhdg  wklv
Ecp  aqw  tgcf  vjku
Dbo  zpv  sfbe  uijt
Can  you  read  this
```

Note that it's better to work with the whole message rather than individual words because occasionally English words ("clear text") show up by mistake. For example, the "word" HTQI backs up to the word "FROG" on the second try, and to the word "COLD" on the fifth.

Decoding a Caesar cipher, simple as it is, stresses the usefulness of the computer and its limitations. It can do the drudgery, but you have to recognize when to stop. For the more complicated ciphers described in what follows, this division of labor is essential.

More complicated ciphers

Since a shift cipher provides virtually no security, the next step is to change the letters in a more random manner. Write down the alphabet and below it write all the letters in some crazy order:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
QAZXSWEDCVFRBGTYHNUJMIKOPL
```

Now every time you see an A in your original message, replace it with a Q, all B's become A's, C's become Z's, and so on. This cipher can't be broken by the techniques used for shift ciphers but it's extremely hard to remember the random alphabet used for the code. So around 1600, in an attempt to combine the virtues of this method with the ease of shift codes, people began to use a key word cipher. The idea is to replace the letters of the alphabet by the letters in the key words, using the order they occur there. For example, suppose my key is: THE RAY GUN ZAPPED ME QUICKLY. Now look at the following:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

and

```
THERAYGUNZPDMQICKLBFJOSVWX
```

What I've done is take the individual letters from the key phrase, avoiding duplicates as needed, and place them below the normal alphabet. Since my key phrase contains only 18 different letters, I've placed the unused letters at the end. To encipher a message using this code, replace the letters in the original message by the ones directly below them—A by T, B by H, and so on.

Here's one possible outline for a procedure that takes a key phrase and creates the code:

```
Get keyphrase
    Run through each letter in key phrase
    Check if already used
        If not used:
        store in next place in cipher list
        mark that letter as used
        until no more letters in the keyphrase
    Now store unused letters from normal alphabet into key.
```

However, this turns out to not quite be the best way of proceeding. For that, you should imagine that what is really going on is a swapping between two alphabets, an encoding one and a decoding one. Suppose you wanted to decipher a message enciphered this way. Say you see an A in the coded message, then, because an A is below an E in the alphabets given above, the original letter must have been an E.

To set up the lists, start with two ordinary alphabets. Now, since T replaces A, we swap the A in the first alphabet with the T in the second. Next we swap the B and the H. How can we tell if a letter is already used? Just look at that letter's position in the second alphabet. If the letter is still there then that letter has not been used. When we are done with the letters in the keyphrase, then any remaining letters should be swapped out of the first alphabet into the second one.

To actually write this program I'll set up two lists. To make my life easier I use two arrays of integers DIMmed to run from 65 to 90, the ASCII codes for A-Z:
So first look at this:

```
DIM EncodeAlph(65 TO 90),DecodeAlph(65 TO 90)
'Now call:
SUB Initialize

SHARED EncodeAlph(),DecodeAlph()

FOR I = 65 TO 90
 EncodeAlph(I) = I
 DecodeAlph(I) = I
NEXT I

END SUB
```

Now I can write the SUB that makes both lists by translating the preceding outline. To do that I need to keep track of where I am in the original alphabet, because that determines where the letter will go. I'll call that PosOfLet. Each time I use a letter from the key, I swap out the letter determined by this position number with its counterpart in the other alphabet, determined from the key, and increase PosOfLet by one. The tricky part comes when I've used up all the letters in the key. Now I have to decide where to put the remaining letters from the first alphabet. The problems come because there is no convenient pointer to the unused letters in the second alphabet. To take care of this, I'll set the used letters to the negative of their ASCII values. For example, if, say, X,Y,Z were the only letters not used in the key, then they would be the only ones that were still positive in the DECODE alphabet.

Here is this SUB:

```
SUB Makelists (Key$)

SHARED EncodeAlph(), DecodeAlph()

LenKey = LEN(Key$)
PosOfLetUsed = 65                 'start with ASC("A")

FOR I = 1 TO LenKey

 A$ = MID$(Key$, I, 1): A$ = UCASE$(A$)

 SELECT CASE A$
   CASE "A" TO "Z"
     A = ASC(A$)
     IF DecodeAlph(A) = A THEN        'character not yet used
       EncodeAlph(PosOfLetUsed) = A
       DecodeAlph(A) = -PosOfLetUsed    'swap the encode/decode
                                        'and flag a used char
       PosOfLetUsed = PosOfLetUsed + 1
     END IF
   CASE ELSE
    ' not a letter - of course you can do something with these to
 END SELECT
NEXT I

' Now throw in unused letters
' This loop should end if either I've used up all 26 letters or
' I can't find any new letters to swap - the X,Y,Z case discussed
' above

FOR I = 65 TO 90               ' start looking in second alph
                               ' here.
   IF DecodeAlph(I) = I THEN
     EncodeAlph(PosOfLetUsed) = I
     DecodeAlph(I) = -PosOfLetUsed    'swap the encode / decode
     PosOfLetUsed = PosOfLetUsed + 1
   END IF

NEXT I

END SUB
```

Now to encode or decode a letter is easy. Suppose you wanted to encode a "C" (ASCII code=67), then we have to look at the value of Encode(67) to find the ASCII for the coded version. Similarly, and this is the nice part, to decode a "C" we just have to look at the absolute value of the entry in Decode(67). Thus we can pass the appropriate array as a parameter, and use this:

```
SUB EncodeDecode(A(),X$)

 X = ASC(UCASE$(X$))
 X$ = CHR$(A(ABS(X)))
END SUB
```

Breaking substitution ciphers

Having spent all of the previous section on a fairly subtle program to create a key word cipher, you might expect it to be secure—or at least difficult to break. Unfortunately, any substitution cipher can be broken, given enough text. In fact, assuming the encoded text was originally written in standard, everyday English, it's pretty easy if you have, say, one thousand words of encoded text.

This section explains how, given enough text, you can break any substitution cipher. All it will require is patience, some paper, and perhaps a program or two that you might want to write. Along the way, you'll learn how to recognize, with some degree of certainty, whether the person used a substitution cipher.

After reading this section, of course, you probably don't want to use a key-word cipher, so I end this chapter with a different ciphering method that is much harder to break. I'd venture to say that most people would find it impossible.

The key to breaking a substitution code is: letters do not exist in isolation. E is almost certainly the most common letter, T is likely to be the next most common, and the frequency of A's is likely to be third highest. Over the years cryptographers have examined thousands of pages of texts to determine the frequency of letters in standard English.

Here's some of what they found, in order of occurrence, in large samples.

I	II	III	IV	V
over 10%	5-10%	2-4.9%	1-1.9%	under 1%
E(13%)	A(8.1%)	L(4%)	G	V
T(10%	O(7.9%)	D(3.8%)	P	K
	N(7%)	F(3%)	Y	X
	I(7%)	C(3%)	W	J
	S(6.5%)	M(2.5%)	B	Q
	R(6.3%)	U(2.4%)		Z
	H(5.4%)			

Some lists reverse a few of the letters: H before R, and D before L for example. Also the percentages are only approximate. What's important is that only in extraordinary circumstances do letters wander out of these groups. For example, somebody once wrote a novel without using the letter "E."

If you have an enciphered text, and it has symbols, which may or may not be letters, and the various symbols occur in approximately these proportions, then you quite likely have a substitution cipher at work.

More is known:

1. A single letter word is A or I or very rarely O.
2. The most common two letter words: OF, TO, IN, IT, BE, AS, AT, SO, WE, HE.
3. The most common three letter words: THE, AND, FOR, WAS, HIS, BUT, YOU, ARE.
4. Four letter words: THAT, HAVE, FROM, THIS, WILL, YOUR.
5. Initial letters of words: T, A, O, S W, I, H, C, B.
6. Final letters: E, S, D, T, N, Y, F, R.

To prevent a cryptographer from using this extremely useful information, it's quite common to eliminate spaces between words, punctuation and use only uppercase letters. After all, YOUCANREADTHISCANTYOU? Actually, since the invention of the telegraph, the custom was to send coded messages in groups of five, for example:IXLPD EXPUT FQDUA—but these groups would rarely correspond to words. The coded message given in the exercises follows this convention.

To replace this hidden information, cryptographers have investigated the most common two and three letter combinations (called bigrams and trigrams).

7. Bigrams: TH, HE, AN, RE, IN, AT, ON, ER, ND, ED, OF, EA, ST, TI, EN
8. Trigrams: THE, AND, ING, ENT, TIO, ION, FOR, NDE, HAS, NCE.

The average vowel count is 40%, and two vowels almost never occur together; when they do, they're likely to be, in order, AI,IE,EI.

To decipher a text encoded with a substitution cipher, first find the frequencies of the letters or symbols. Then substitute the letters using the frequency tables. If your text is long enough, then you're probably done at this point—you should be able to read the raw text. For example, even if a few letters are wrong you'll get something like:

TOYEORNOTTOYETHATIDTHEKZEDTION

but you can understand this. Here, of course, any program you write will only get you to this point, but then your intuition must take over.

If the text isn't long enough, then some of the frequencies will be off, and you'll have to try to make changes, but usually you'll be able to keep within the groups.

At the very least any program you write needs procedures to:

1. Analyze the frequency of the letters.
2. Analyze the frequency of the bigrams.
3. Display the best guess for the text.
4. At any point allow you to enter changes, and using this newly modified key, display the text using this modified key.

5. Display the current key.
6. A practical cipher.

The problem with a simple substitution cipher is that, by always replacing a letter by the same symbol, someone can break it using frequency analysis. One way to avoid this method of breaking a code is to change the substitution. Instead of always replacing, say, an E by the letter T, use a T the first time, a Z the next. So each time an E occurs it is replaced by another letter. This method is called a multi-alphabet substitution cipher. It's much more difficult to break this cipher, but it's also much more difficult to set up. After all, you have to devise a way of getting these multiple alphabets.

I'll use the built-in random number generator in QB to generate my alphabets. Recall that the command:

```
X = RND(1)
```

gives you a different "random" number between 0 and 1. However, and this is the key to breaking the cipher, I'm going to show you: given enough data, a professional cryptographer will find out the next number in the sequence. A cryptographer would say that the random number generator in QuickBASIC isn't cryptographically secure. Finding a cryptographically secure random number generator is probably the most important problem in cryptography. The ones cryptographers think are best use the factors of large numbers. But even these have not been proven to be secure—all that has been proved is that if factoring is hard, then they are secure. So they ultimately depend on nobody ever discovering a fast method for factoring. Most mathematicians, including myself, believe this, but this is a far cry from proving it.

Anyway, the idea for the cipher that follows is that we scale this number and use it to determine the Caesar shift. Now instead of using the same shift for the next letter, use the random number generator to get a different shift for each letter. Each time you encode a letter it's transformed differently.

Unfortunately, this method won't quite work. Since the patterns don't obviously repeat, there isn't any reasonable method of decoding the message. You would never know what to shift back by. You have to modify this idea slightly. For this, recall that QB has a built-in debugging tool for dealing with programs that use random numbers. If you use:

```
RANDOMIZE Seed
```

then QB always gives you the same sequence of random numbers.

Each seed gives a different, repeatable sequence of random numbers. Ask the person for a key, say a four digit number. Use this to re-seed the random number generator:

```
RANDOMIZE Key
```

Now you can generate a list of shifts, one for each character in the file:

```
NextShift = INT(256*RND(1))
```

Use these shifts just like a Caesar cipher—CALL this shift generator for each letter in the message.

How to decode: The whole point of repeatability is that if you process the command

```
RANDOMIZE Key
```

again, then, when you generate Caesar shifts, you get the same series of numbers you did before. And just as before, if you know what the original shift was, you can reverse it just as easily.

Unfortunately, this doesn't quite work in the current version of QB. The RANDOMIZE Key doesn't work as announced. Instead, you have to say X = RND(negative number) to get the repeatable sequence.

In any case I'll leave it to you to implement this, because there's a much more elegant way of proceeding. Recall from the bit twiddling section of the last chapter that the XOR command has the nice property that if:

```
B = A XOR Shift
```

and you do it again:

```
C = B XOR Shift
```

then the value of C is the value of A. Thus by XORing twice you get back to where you started. This means you can use the same module to both encode and decode:

```
SUB EncodeDecode (filename$, keyvalue)

DIM SingleChar AS STRING * 1          'for use in GET and PUT
X = RND(-keyvalue)
FileNum = FREEFILE
OPEN filename$ FOR BINARY AS #FileNum

FOR I = 1 TO LOF(FileNum)
     GET #FileNum, I, SingleChar
     CharNum = ASC(SingleChar)
     RandomInteger = INT(256 * RND(1))
     CharNum = CharNum XOR RandomInteger 'this is it
     SingleChar = CHR$(CharNum)
     PUT #FileNum, I, SingleChar
NEXT I
CLOSE #FileNum

END SUB
```

11
CHAPTER
Finishing up

This chapter shows you some advanced topics that, for one reason or another, haven't found a home elsewhere in this book. The next to last section gives some hints on moving on to the amazing Microsoft Professional development system—which I think is only necessary for commercial BASIC programming. The final section gives some bibliographic hints.

You'll see:

1. More on full menus.
2. Managing multiple module programs.
3. Using libraries, including commercially available ones.
4. Using DOS interrupts.
5. Bit twiddling: Peek, Pokes, DEF SEG, BLOAD and BSAVE.
6. Multiple screen pages.
7. Event trapping.
8. Using a Mouse.

Using full menus

Except for a brief use of full menus way back in chapter 1, I've stuck with easy menus. Now seems like a good time to explain the differences.

I'll briefly describe some of the full menu options in this section and the next.

The *Learning to Use Microsoft QuickBASIC* manual contains more information, if you need it.

Go to the Options menu and toggle Full menus on. You've already seen in chapter 1 that the full files menu lets you turn syntax checking off. Once it's off, then you can manipulate ASCII file. More importantly, without using the full files menu, you can't effectively create large programs (see the next section).

Besides syntax checking, the only change in the full Options menu is a command that affects a mouse. Right Mouse lets you change what clicking the right mouse button does. Normally, it gives you context sensitive help (=F1). You can change this to "Execute up to this line," which means that QB will run the program from the beginning to where the mouse cursor (■) is. This is the same as moving the ordinary cursor (_) to the same place and hitting F7.

The next menu, Calls, is occasionally useful when debugging, especially for recursive programs. After any break it shows you the current nest of procedure calls needed to get to the breakpoint. If your program doesn't seem to be calling the right procedures, then this can be a lifesaver. The Calls menu is like a stack of procedure calls. The top item gives you the name of the procedure where the program broke; the second line is the name of the procedure that called the first procedure; the third line called the second line. The last line in the Calls menu is likely to be the name of the main module.

The Debug menu has quite a few new options in full menus. Working from the top down, the first new option that you'll see is:

```
Watchpoint . . .
```

Choosing this option opens a dialog box in which you set up a condition for the program to break. For example, you can enter I=10 and then the program would break when the value of that variable, in the module where you set the watchpoint, reaches 10. My favorite use of watchpoints is to combine them with a STATIC variable. I'll set up a static variable in a procedure, add one to it each time the procedure is called, and use a Watchpoint to break after this static variable reaches, say, 5. Then I can check what's happening after five procedure calls; to check every other time, I can break on S MOD 2 = 0. I find this very useful when debugging recursive procedures.

"Delete All Watch" is a shortcut for eliminating all the entries in the watch window.

"Trace On" runs your program in slow motion. It can also be turned on with the command TRON (TRace ON) within a program; TROFF turns it off in a program. I never find this useful, because the switch back and forth to the output screen is usually too fast.

"History On" lets you record the last 20 lines of the program. Usually, I'll use this after a break to examine the fine structure of a program. For example, suppose you get a run time error. Turn History on, and re-run the program. Now

QB keeps track of the last 20 lines it executed prior to the run time error. You can now step both backwards (Shift+F8) or forwards (Shift+F10). You can only step forward after you've stepped backwards. When trace is on, QB also records the last 20 lines. The "Break on Errors" is used when a program uses error trapping (see the last section).

You'll see more about the full Run menu in the next few sections. For now, I just want to point out that the Make eXe File now has another option in its dialog box. You can create an .EXE file that "requires BRUN45.EXE." Previously, a stand alone file really could stand alone. QB added all the routines that it needed when it made the .EXE file. This made the .EXE file for even the smallest program at least 14,000 bytes. By creating a program with the "requires BRUN45.EXE" option, you leave out the support routines from the .EXE file.

Of course, QB will then need the BRUN45.EXE file to actually run the program. However, if you have 10 .EXE programs on a disk, then you'll only need one copy of BRUN45.EXE on the same disk or accessible via the PATH command, so the savings are considerable. For commercial programs, you'll generally not want to use this option, since programs will run slightly faster if they're compiled without this option.

The Search menu has two new options. Selected text (shortcut Ctrl+ \) lets you search for the next occurrence of text that you've selected in the active window. It's useful, for example, if you want to find the next occurrence of a variable. The Label command searches for line labels by adding a colon to whatever you enter in the dialog box and looking for that.

The View menu adds four choices. Next Sub is just Shift+F2. It moves the next (alphabetically) procedure to the active window. The Split command is a toggle. It allows you to have two parts of a program on the screen at the same time. This is occasionally useful if you need to see, say, the main module, while editing in a procedure. Combine it with the Alt+ combination, and you can have each window about 10 lines wide. Only one of the possible five windows can be active at any one time, and you still cycle through them with F6 or Shift+F6. The Next statement moves the cursor to the next statement to be executed. (Especially if you've "broken" a program, moved the cursor around and forgotten where the break was.) For the Included lines and Files command, see the next section.

All the new options on the full Edit menu have shortcuts. Clear is pressing the Delete key after text is selected. New Sub, New Function are done simply by typing the appropriate keyword and hitting Enter.

Most of the options on the full file menu will be described in the next section. The only one I want to mention here is the DOS Shell option. This lets you temporarily exit to DOS, so you'll be able to run any .EXE or .COM file. When you're done, type EXIT to return to QuickBASIC. Two points to be aware of: SHELL doesn't always work with DOS 2.0 or 2.1—you're best off with DOS 3.0 or a later version. Next, SHELL loads another copy of COMMAND.COM and the program you'll be running. In particular if you don't have at least 512K, be very careful—your system may lock up.

Managing multiple module programs

I've mentioned before that you'll likely want to reuse procedures and functions. The easiest way to do this is with the Merge option on the full file menu. This inserts a text file stored on a disk at the cursor's current location. When you choose it, QB presents you with a dialog box, similar to the one for Open program. Only this time, the file you choose doesn't replace the one in memory, it's merged with the one in memory. (Warning: you cannot merge any information that has been stored in QB's "quick load" form—make sure that you've chosen "Text—readable by Other Programs" option when you first saved it.) Although merging doesn't change the stored version, it does make the current program that much larger. When you save the program it will now include a copy of the merged text. The only question is, what's the best way to store a procedure/function, or group of procedures and functions, as a text file on a disk. Obviously, it may be code that you've bought, borrowed or written before. In this case it will be stored on a disk with its own name.

It turns out, however, that you can write a complicated program with the pieces considered and stored as separate files. Essentially, then, you'll be able to tie separate pieces (modules) into a program, and have QB do the bookkeeping. You'll be able to save these pieces as separate files—so sometime later you can, if you want to, physically merge them into larger programs.

To create a multiple module program from scratch, use the Create File command on the full File menu. We used this in chapter 2 to create a document. Now, I want to say more about the other two choices in this dialog box. The first and most important of these two options is "Create as Module." If you enter a name in this dialog box, then it will be listed separately in the SUBs (F2) list. The SUBs list indicates if a module has its own procedures and functions, by indenting its name from the name of its main module.

Within a QB program, you can create as many modules as fit into RAM. And, so a QB program may consist of only one module (all the ones so far in this book), or many modules. QB automatically keeps track of what modules are used in a given program, via a file usually called a .MAK file.

For example, suppose that your program is called LONGPROG.BAS. In the course of writing this you create four modules called: SHORT1,SHORT2, SHORT3 and SHORT4. Then when you save LONGPROG.BAS, QB automatically creates an ASCII file called: LONGPROG.MAK which consists of four lines of text:

```
LONGPROG.BAS
SHORT1.BAS
```

```
SHORT2.BAS
SHORT3.BAS
SHORT4.BAS
```

Obviously, if you delete or change the .MAK file, then QB will no longer be able to recreate the source code of your program.

The next option in the Create file dialog box is Include. An Include file is a text file that you can put into your programs via the $INCLUDE meta-statement. A statement like:

```
' $INCLUDE "SHORT1.INC" (or REM $INCLUDE "SHORT1.INC")
```

has QB search for a file called SHORT1.INC; where it searches depends on the path options that you've set, c.f. the options menu. When it finds it, then it will physically merge it into the source code for your program at that point. INCLUDE files usually consist of DECLARE statements and TYPE definitions used for the given module. After all, suppose you want to CALL a procedure, or use a function in one module that is defined in another module. Then, without an INCLUDE file in the first module that contains the appropriate DECLARE statements for the second module, QB would not know how to use them, what types of parameters can be passed, etc. Since the whole point of multiple module programs is to allow you to use the procedures and functions defined in all the modules, you can see the importance of these kind of include files.

There are many ways to create this type of $INCLUDE file for a given module. The easiest is to group the TYPE . . . . END TYPE statements and the DECLARE statements in one place. Most often you'll use QB's automatic generation of DECLARE statements to create the DECLARE statements. Then edit out all the functions and procedures. Finally, save the remaining code using the SAVE AS command on the file menu to create the include file.

The View menu contains two options for handling INCLUDE files. Suppose the cursor is on a line containing a $INCLUDE meta-command. Then Included File lets you edit the INCLUDE file, Included Lines lets you look at, but not edit, the Include file.

The Load File command opens a dialog box that gives you the same three options as Create File. You can load a text file as a program module, include file, or document. Once you have created a multiple module program, then the Save All option lets you save all the modules. On the other hand, the Unload file option opens a dialog box that lets you remove modules from a multiple module program; the names will also be deleted from the .MAK file.

Finally, I should mention that it's possible to use SHARED variables in a multiple module program—but I think it's never a good idea. If you're interested, then look at the discussion of the COMMON command in the Programming in BASIC manual. Instead, I try to make modules as self contained as possible, commenting them copiously, of course. And I only use parameters to communicate between multiple module programs. Finally, the SUB's option (=F2) lets you move a procedure from one module to another.

Libraries

A quick library is best thought of as commands that you are temporarily adding to QuickBASIC. It's made up of pre-compiled procedures and functions, possibly created in another programming language, that you can use like any other QB statement. Also, since quick libraries are not readable, they give you a way to protect your source code from other people's eyes.

You tell QB to load a library by invoking QB with the /L option, followed by the name of the library. The custom is to use .QLB as the extension for quick libraries. You can only use one library at a time. If you don't specify a library, but do use the /L option, then QB loads the default library QB.QLB that comes with QuickBASIC. The QB.QLB library has three precompiled assembly language programs that will let you, for example, find the amount of free space on a disk in an efficient way (see the next section).

A quick library made from QuickBASIC source code is most easily created from within QB. Use the editing facilities of QB to remove any procedures or modules that you do not want as part of the library. However, be careful; unlike a module, a library can only use procedures and functions from within itself. Also, it's best to strip out any unessential module level code before creating the library. If you want your library to contain another library, then you'll need to have invoked QB with the /L option, giving it the name of the smaller library. To actually create the library, choose the Make Library from the Run menu. This opens a dialog box that you fill in the name of the library. If you don't give an extension for it, then QB uses .QLB.

Quick libraries are basically standalone programs without support routines. As such they run very fast. However, you no longer can tell at a glance what are the names of the procedures and functions that are included in the library. Since you can't use the same procedure name twice, you'll occasionally want to examine a quick library with the program QLBDUMP.BAS that is supplied with QuickBASIC. This program can analyze any quick library and list all the procedures, functions and data contained in the binary file that makes up the library.

For more on creating libraries via LINKing and the standalone BC compiler, see appendix G and H of the Programming in BASIC manual. These sections will explain what's necessary for creating "mixed language libraries."

Finally, commercial developers will supply you with libraries that improve QuickBASIC—they'll either give you new functions, or faster versions of functions that you may have written yourself. Check out Byte or one of the other PC magazines for more details.

Bit twiddling and interrupts

A computer is ultimately a giant collection of on-off switches, and a disk is a collection of particles that can either be magnetized or not. Think of each memory location in your PC as being made up of eight on-off switches. This affects the internal representation of numbers inside a PC. For example, when you write 255 in decimal notation, you mean two hundreds, five tens and five ones. The digits are arranged in positional notation with each place holding numbers ten times as large as the one to the right. Your computer would store this number in a single memory location as:

11111111

meaning one 128, one 64, one 32, one 16, one 8, one 4, one 2 and one 1. One way to indicate that a number is a binary number in print is to use a subscript of 2; use a subscript of 10 for decimal numbers. Each of these switches is called a bit, for binary digit, and when it stores a 1 we say the bit is on, and the value stored in that position is 2^i ($=2^i$). In binary notation each place holds numbers twice as large as the one to the right. Eight bits form a byte (= one memory location), two bytes form a word, and 1/2 a byte is called a nibble.

Here's how you count to 15 in binary:

binary	**decimal**
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

15 is the largest number that can be stored in a single nibble, and 255 is the largest number that can be stored in a byte. Bits are numbered with the left most called the most significant and the right most, or 0th, bit the least significant.

You may have thought that QB's using 0 for false was quite natural, but why -1 for its internal representation of TRUE? To understand this you first have to know that, ultimately, all the logical operators (NOT, AND, OR, XOR) work on the bit level. For each of the operators, QB looks at the binary digits (bits) one at a time, and follows these rules.

AND makes the binary digit 1 only if both bits are one, otherwise it is 0. For example, if:

$X = 12_{10} = 1100_2$ and $Y = 7^1{}_0 = 0111_2$

then X AND Y $= 0100_2$ (=4), because only in the third position are both bits one. Because AND gives a one only if both X and Y have a 1 in that place, by ANDing with a number whose binary digit is a single 1 and all the rest of the bits are 0, you can isolate the binary digits of any number. This is called *masking*.

OR on the other hand gives a 1 if either of the binary digits is 1. So X OR Y $= 1111_2$ (=15). Use OR to make sure that specific bytes of a variable are on. For example:

```
X = X OR 64
```

makes sure that the fourth bit in X is on. X = X OR 96 turns on the third and fourth bit. Remember, bits are numbered starting from 0. You can try all these in the Immediate window via, for example: X= 12:Y =7: PRINT X OR Y.

One of the most interesting operators is XOR, which gives a one in a specific position if exactly one of the digits of X and Y in that position is one. So X XOR Y = 1011 ($=11_{10}$). XOR has the useful property that XORing twice doesn't do anything; (X XOR Y) XOR Y = X.

There are two other operators: IMP and EQV. The former is, 1 except when X has a 1 and Y has a 0, and the latter is 1 when they are both the same, either both true or both false.

NOT on the other hand, works on a single number by reversing the bits—a 1 becomes a 0, and a 0 becomes a 1.

Now why is -1 = TRUE? First off, QB stores each integer as a single word using only 15 bits. So $0 = 000\ 0000\ 0000\ 000_2$. As you'll see grouping by fours in binary is more useful than grouping by threes as in done in decimals. Anyway, NOT 0 is:

1111 1111 1111 1111

You might expect this to be the largest integer expressible in 16 bits, which is 65,535 in decimal. However, because QuickBASIC uses only 15 bits for its integers, it uses the left most bit for the sign. A one in the left-most bit means the number is negative. Moreover, it uses what is called twos-complement notation for negative numbers. In twos-complement notation to represent a negative number, $-X$, you (1) apply NOT to the 15 bits that give X, (2) set the left-most (16th) bit to one, and (3) add one to the result. So for −1, you take the bit pattern for 1:

000 0000 0000 0001

apply NOT:

111 1111 1111 1110

add a 1 to the left most bit:

1111 1111 1111 1110

and finally add 1. The result is that:

1111 1111 1111 1111 = NOT(0)

is −1. For the reasons why this system is so useful, look at a book on microcomputer architecture like S. Morse's *8088 Primer*, (Howard Sams).

One place where you need binary digits is when you want the LINE command to draw a dotted rather than a solid line. This is done by adding a final option to the LINE command:

```
LINE (x1,y1) −(x2,y2),[Color],[B[F] ],style
```

This style option is a number or numeric expression that controls the form of the line. The way style works is that you should think of a line as a group of 16 pixels, repeated as needed. To turn a specific pixel off, make that binary digit of the number in the style position a 0. For example, if you want every other pixel to be off you can use a number which is the decimal equivalent of the binary: 0101 0101 0101 0101 = 30583 (= 1 + 2^2 + 2^4 + 2^6 + 2^8 + 2^10 + 2^12 + 2^14).

Binary numbers are obviously a bit difficult for people to handle. It's much easier if you use hexadecimal numbers (base 16). In this case, hexadecimal 10 ($=10_{16}$) is 16 in decimal. You use A for decimal 10, B for decimal 11, C for decimal 12, D for decimal 13, E for decimal 14 and F for decimal 15. Each hexadecimal digit represents 4 binary digits, or one nibble. To convert binary numbers to hex (shorthand for hexadecimal) just group the digits from right to left in groups of 4, and convert. So:

11010111 (1101 0111 in two groups of four)

is D7 in hex because 1101 = 13 in decimal = D in hex, and 0111 is 7 in both decimal and hex. You can use hex numbers in QB by prefixing them with an &H. So 49 = &H, or PRINT &HF + &HF would give a 30 on the screen; displays are always in decimal. Hex notation makes deciding on the numbers for LINE styles easy: change the pattern you want into 4 hexadecimal digits.

Moreover, QuickBASIC has a built-in function to convert a decimal number to a string containing the hexadecimal digits. It's HEX$(numeric expression). So

HEX$(32767) = "7FFF." There's no built-in equivalent to VAL—you'll have to write one yourself.

One of the most important uses of bit level operations in QuickBASIC is through the DOS Interrupts. An interrupt forces QB to suspend what it's doing, and to have DOS tell the central processing unit (CPU) to do something else. To use a DOS interrupt, you have to invoke QB with the /L option, to load the QB.QLB library. Next, you set up a user-defined type:

```
TYPE CPURegisters
    AX AS INTEGER
    BX AS INTEGER
    CX AS INTEGER
    DX AS INTEGER
    BP AS INTEGER
    SI AS INTEGER
    DI AS INTEGER
    Flags AS INTEGER
END TYPE
```

These correspond to the eight registers inside the CPU. Now DIM two variables as being of this type:

```
DIM InReg AS CPURegisters, OutReg AS CPURegisters
```

Each register is divided into two nibbles. This is where you need binary operations; you need to read, turn on and turn off individual bytes inside individual registers. For example, the AX (accumulator), like all the registers, is divided into two nibbles, called AH ("A high") and AL ("A low"). To put a number, say, decimal 54, in the AH component of InReg.AX, use:

```
InReg.AX = 54*256
```

The 256 shifts the bits to the high order nibble. You could also directly use hex notation to make the low order nibble 0, and the high order nibble decimal 54 (=&H36), by setting Inreg.AX = &H360000.

Roughly speaking, what you now have to do is put certain numbers in the components of the variable InReg, and read off the results from the components of OutReg. Assuming you have loaded the QB.QLB library, you can use the:

```
CALL INTERRUPT(interrupt number,inreg, outreg)
```

command that QB.QLB adds to QuickBASIC, and then read the results from OutReg.

The most important interrupt number is decimal 33 (&H21). Among its many other powers, this will let you efficiently find the amount of free disk space. To do this you:

1. Place the number of the drive in the InReg.DX component (0 is the current drive, 1 is A:, 2 is B:, etc.)
2. Place a 54 decimal in the AH part of InReg.AX. (This is because 54 (=&H36) is the DOS function for "Get Free Disk free Space")
3. CALL INTERRUPT (33,InReg,OutReg)

or, equivalently:

3a. CALL INTERRUPT (&H21,InReg,OutReg)

Now the amount of free disk space is obtained by multiplying the AX,BX and CX components of OutReg. If there's an error—like the disk drive doesn't exist, then OutReg.AX = &HFFFF.

As another example, suppose you want to hide a file; make it so it doesn't display after use of the DIR command. This is done with the &H43 function of INT &21H. First off, you need to tell DOS the name of the file. DOS doesn't use ordinary ASCII strings for file names; instead it uses what are called ASCIIZ strings. This is an ordinary ASCII string with a CHR$(0) attached at the end. Next, you need to tell the &H21 interrupt where in memory it can find this name. (After all, registers contain only numbers.) For this, and the rest of the function, see the next section.

Here's a list of some of the other common functions available through the $H21 interrupt:

Function &H0E: sets the default drive. Use this by setting AH = &H0E, and placing the appropriate integer in DL, 0 for A, 1 for B, etc.

Function &H0F: Gets the current type of display. Except for the problem with Hercules cards, this would be a better way than using the error handler of chapter 8. After you set AH = &H0F, and call this function:

AH will give you the number of columns, and

AL =	&H00 40×25 monochrome text for CGA
	&H01 40×25 color text for CGA
	&H02 80×25 monochrome text for CGA
	&H03 80×25 color text for CGA
	&H04 320×200 4-color graphics
	&H05 As in &H04 with color burst off
	&H06 640×200 2 color graphics
	&H07 monochrome text only
	&H08-&H0A The late lamented PCjr.
	&H0D 320×200 16 colors EGA
	&H0E 640×200 16 colors EGA
	&H0F 640×350 monochrome graphics on EGA
	&H10 640×350 either 4 or 16 colors on EGA

&H11 monochrome graphics (MCGA,VGA)
&H12 640×480 16 colors on VGA
&H13 320×200 256 colors on MCGA or VGA

Function &H19: Gets the default drive. On return AL contains the code for the drive (codes as above).

Function &H21: Finds the version of DOS being used. On return AL contains the major version number, i.e., DOS 3.3 would give a "3," and AH the decimal part.

For more on registers or the DOS interrupts, see one of the books mentioned in the bibliography.

DEF SEG, BLOAD and BSAVE

As well as putting information directly into registers you can also examine, save or place information in specific memory locations. While you rarely need to be concerned about this, occasionally you have to go to this level. For example, having spent some time drawing a complicated graphics object, it would be nice to save it in such a way that you can reproduce it on the screen without having to do the calculations again.

The command PEEK is the one to look at a specific memory location, and the command POKE puts information at a specific location. The command BSAVE saves the information in memory, and the command BLOAD puts information saved via BSAVE back into memory. To use these commands you must first learn how to refer to memory locations. The original 8088/8086 had only twenty signal pins coming out of it. This meant it could refer to only 1 megabyte (1,048,576) locations. Unfortunately, the internal architecture (i.e., the registers) in the CPU could handle only 16 bytes of information at a time. Sixteen bytes gives you only 65,536 (64K) locations, so a method was devised to divide memory into groups of 64K, each one of these groups called a segment. Next, to allow easy communication between the segments, they were made to overlap to the maximum extent possible. Since you're allowed 65,536 segments numbered from 0 to 65535 or &H0 to &HFFFF, a little math shows you that each segment starts a mere 16 bytes in from the previous one. The number of bytes you've moved from the start of each segment is called the offset, and you also use 0 to 65535 decimal (or &H0 to &HFFFF) for them. The convention is to use the following notation for memory locations:

```
segment#:offset
```

For example, 0:33 refers to the 33rd memory location; this is sometimes called the 33rd absolute location. Obviously, a memory location can now have many ways of addressing it. For example, I could have used 1:17 or 2:1, because the first segment starts at absolute memory location 16, and the second segment starts at absolute memory location 32.

The command DEF SEG = N (0 < = N < = 65535) sets the segment for all the memory commands that will follow. For example, the medium resolution screen is stored in segment 47104, from offset 0 to offset 16383 (= &H4000). Now, to save a screen image you can use:

```
DEF SEG 47104     (or DEF SEG &HB800)
BSAVE "filename",0,&H4000
```

to recall the screen use:

```
DEF SEG 47104     (or DEF SEG 47104)
BLOAD "filename",0
```

In general the second slot in the BSAVE, BLOAD command holds the offset from the segment defined by the last DEF SEG. (To restore the segment to the 0th, use DEF SEG 0 or DEF SEG.)

You can also look at a specific location in the DEFined SEGment by the command PEEK(offset), and actually place a specific value via POKE(offset). Obviously, be very careful of POKEing values into memory—it's easy to crash your machine if you overwrite one crucial byte.

As I mentioned in the last section, to use one of the interrupts that affect files, you need to give DOS the location in memory where that variable is stored. To find the segment address of a variable, use the function VARSEG, to find the offset, use the SADD function. For example:

```
ASCIIZ$ = FileName$ + CHR$(0)
InReg.DS = SSEG(ASCIIZ$)
InReg.DX = SADD(ASCIIZ$)
```

You can now use the &H43 function of interrupt &H21 to read or change the attributes of FileName$, as follows:

```
See AH = &H43
    AL = 0 to read the attribute
    AL = 1 to set the attribute
Now if AL = 1, then the contents of CX determine what happens:
    CX = 0 – changes file to read-only
       = 1 – hides files
       = 5 – changes the archive bit (used for backups)
If this function fails, then AX is non-zero.
```

Multiple screen pages

Occasionally, you'll want to temporarily overwrite or erase the screen, but to now you haven't seen a good way to go back to its original form. For example, the last section of this chapter talks about error trapping. As you can imagine, this will occasionally involve writing messages on the screen asking you to correct the error so that you can continue. But, after you correct the error, how do you get back to the original screen?

The easiest way to do this is to use the multiple screen pages that your PC provides in text mode. Depending on your adapter, you'll also have multiple pages in the various graphics modes as well. Consult the manual and the help screens for this. For example, you can write on one text screen while displaying the other, and then switch back and forth between them. This is done by a modification of the SCREEN command.

The full form of the SCREEN command takes four options:

```
SCREEN , , , ,
```

The first two options you've already seen. Recall that the first slot is for the screen mode, and the second is not zero when you want color to be displayed on a composite monitor. The last two slots are the ones that concern us now. The third slot holds the number of the page that you're currently writing to, and the fourth holds the page that's currently visible. In 80 column text mode (SCREEN 0:WIDTH 80) you have 4 pages numbered 0 to 3 for these slots. A fragment like:

```
SCREEN 0, , 1, 0
CALL WriteMessage
SCREEN 0, , 1, 1
```

will call a procedure to write a message on the next text page, while keeping the original text screen intact. The:

```
SCREEN 0, , 1, 1
```

command will display the new message. Finally, you can use:

```
SCREEN 0, , 0, 0
```

to go back to displaying, and writing, on the original screen.

QuickBASIC also makes it easy to copy the contents of one screen page to another. The command:

```
PCOPY frompage, topage
```

copies the entire contents of one screen page to another. One common use of PCOPY is when you want to preserve the original page intact, but still use part of it in the second page.

Subroutines and event trapping

A subroutine can be thought of as a less powerful form of procedure, that QuickBASIC inherited from interpreted BASIC. It would be nice if it wasn't needed anymore—unfortunately it is, occasionally—you'll see why in this section and the next.

Interpreted BASIC doesn't have procedures; instead you have to use subroutines. In QuickBASIC, a subroutine is a group of lines that begins with a label or a line number, and ends with the command RETURN. (Since interpreted BASIC doesn't have labels, there they would be just a group of lines starting at a given line number.) Just as the CALL command transfers processing to a procedure, the command GOSUB transfers processing to a subroutine. QuickBASIC jumps to the statement identified by the label or line number following the keyword GOSUB. So the command GOSUB ThisIsALabel would transfer processing to the statement after the label ThisIsALabel:—you don't use the colon when transferring control to a labeled routine. Similarly GOSUB 5000 would transfer processing to the statement on line number 5000. QB then processes whatever code appears until it processes a RETURN command. At that point it resumes processing at the statement following the GOSUB command.

The problem with subroutines, and why programmers usually prefer procedures, is that, if you stick with subroutines, then you lose the ability to distinguish between local and global variables. After the command GOSUB, then whatever happens in the group of lines that make up the subroutine is global (visible) to the entire program. In particular, any changes you make to variables will have side effects that may be uncontrolled. Furthermore, subroutines do not allow parameters, so communicating information to and from a subroutine is always messy. For example, consider the procedure SUB StringSpaces(X$), rewritten as a subroutine:

```
StripSpaces:
    Temp$ = " "
    FOR I = 1 TO LEN(X$)
        IF MID$(X$,I,1) < > CHR$(32) THEN Temp$ = Temp$ + MID$(X$,I,1)
    NEXT I
RETURN
```

To strip the spaces off of the value of a variable Phrase$ you need to say:

```
X$ = Phrase$
GOSUB StripSpace
Phrase$ = Temp$
```

The procedure is cleaner—and you don't have to worry about unintentional side effects because of the variables I and Temp$. Stick to procedures, and you know that side effects only come from parameters and shared variables.

Subroutines have another disadvantage: they are not islands of code, and you can inadvertently fall into a subroutine. This means that you must have an END command or some other barrier to separate the subroutines from the rest of the program.

Since subroutines have few advantages over procedures, one would hope that they could cheerfully be eliminated from most of your QuickBASIC programs. Unfortunately, there are some situations where a QuickBASIC programmer must use subroutines. The first comes because interpreted BASIC programs had to use subroutines. It's sometimes easier when modifying an interpreted BASIC program to continue using some of its subroutines.

However, here I want to concentrate on the few times you need subroutines when writing a QuickBASIC program from scratch. For example, suppose you wanted to write a program that, like QuickBASIC itself, gives a user "help" when he or she presses the F1 key. This means the program must monitor the keyboard at all times, and, if the F1 key is pressed, interrupt the normal flow of processing and jump to the HELP routine. You start the monitoring process, in computerese event trapping, by the command:

```
KEY(1) ON
```

After this command QuickBASIC checks between each statement that is processed whether the first function key was pressed. To test the second function key, use KEY(2) ON, etc. The command KEY(1) OFF turns off the monitoring. Obviously, this slows any program down. Therefore, when doing something that makes heavy use of the central processor, like sorting lists, you might want to turn event trapping off. This is done by the KEY(1) OFF.

Having enabled event trapping you have to tell the compiler what to do. This is done with the statement:

```
ON KEY(1) GOSUB . . . .
```

where the dots stand for the label or line number of the subroutine that you want to transfer control to. Nothing prevents you from GOSUBing to different places at different times in response to F1, or trapping more than one function key at the same time. (Use KEY(2) ON, ON KEY(2) GOSUB . . ., etc.) A new ON KEY( ) GOSUB overrides a previous one. Unfortunately, in the current version of QuickBASIC, event trapping can only direct flow to a subroutine, not a procedure.

Sometimes you want to temporarily turn off event trapping, but have the compiler remember that the event happened. This is done with the command:

```
KEY(n) STOP
```

Once this command is processed, the compiler doesn't respond to the event until you turn event trapping back on, with the KEY(n) ON. At that point it will do what it has to do. On the other hand, the command KEY(n) OFF, mentioned before, turns off event trapping, and so eliminates any memories. When QB is inside a subroutine called by a KEY statement then it still keeps track of keystrokes. In effect, the GOSUB causes a KEY( ) stop to be executed as well.

You can also trap time, or a joystick. For example, the command TIMER ON allows you to use:

```
ON TIMER(n) GOSUB . . .
```

to transfer control to a subroutine every n seconds—here n is an integer from 1 to 86,400 (24 hours). While you RETURN from the subroutine the clock is reset to 0. Similarly, TIMER OFF turns this type of event trapping off, and TIMER STOP takes it off the hook temporarily—it keeps track of the time but doesn't do anything—yet.

QuickBASIC's extraordinary power even allows it to trap keys other than the function keys, unlike interpreted BASIC. You can trap as many as 25 different keys at once. In addition to the 10 (or 12, if you have an extended keyboard) function keys, you can trap the four cursor keys. This is done via:

KEY(11) ON	to trap the up arrow
KEY(12) ON	to trap the left arrow
KEY(13) ON	to trap the right arrow
KEY(14) ON	to trap the down arrow
KEY(30) ON	traps the F11 key
KEY(31) ON	traps the F12 key

Moreover you can trap any other 11 keys, or combinations, like Ctrl+C, of your choice. Doing this is a bit tricky—here's the idea: the command:

KEY n,CHR$(state of keyboard) + CHR$(extended scan code) tells the compiler to associate for future event trapping a certain key or combination of keys. The state of the keyboard is any combination of:

The right shift key	(code 1 = &H1)
The left shift key	(code 2 = &H2)
either shift	(code 3 = &H3)
The control key	(code 4 = &H4)
The alt key	(code 8 = &H8)
The num lock key	(code 32 = &H20)
The caps lock key	(code 64 = &H40)

You add these code numbers together to trap the combinations. For example, to trap both Ctrl+Shift, use a 6 (2+4) inside the first CHR$.

The extended scan codes may be found in appendix F. For example, a C has a scan code 46 (=&H2E). So the command:

```
KEY 18,CHR$(7) + CHR$(46)
```

when followed by a KEY(18) ON traps a Ctrl+C.

You can prevent Ctrl+Break, or even a system reset (Ctrl+Alt+Del) from interfering with a program by trapping as well. For example, the extended code for the break key is &H46, so

```
KEY 15, CHR$(4) + CHR$(46)
```

sets up subroutine for Ctrl+Break.

Although you must have the subroutine for the event trap in the module level code, you can place the commands to start the process anywhere. In fact, once an event trap is set up, QB will start searching through the module level code, but if it doesn't find the appropriate subroutine there, then it will search through all the module level code to find that subroutine.

Mice

Personally, I find mice less than overwhelming. A cynical friend of mine once commented that "Mice are for people who think that 'point and grunt' is the best we have to offer users." But I know that he and I are in the minority, so let me say a few words on how you program a Microsoft compatible mouse, using QB.

As you'd expect, you handle mice through a DOS interrupt. This interrupt (&33H) is only available after you install a driver program, usually called MOUSE.SYS. You specify the function by what you place in the AX-DX registers. Here's a list of some of what you can do:

FUNCTION	What to set	What you get back

Reset Mouse	AX = 0	BX = # of Buttons
		IF AX = −1 then no mouse.

Show Mouse	AX = 1	
Hide Mouse	AX = 2	
Get mouse status	AX = 3	BX = Button status = 0 no button down = 1 left button = 2 right button = 3 both (middle)

Button released?	AX = 6
	BX = 0,1,2
		AX = 0 if up
		1 if down

Function 6 also returns in CX the virtual horizontal position and in DX the virtual vertical position. What this means is that regardless of screen mode, the mouse uses a 640×200 screen.

For more details on mice, check out the *Microsoft Mouse: Programmer's Reference*, from Microsoft press.

The professional development system

The professional development system (PDS) in its latest incarnation (7.1) weighs in at a mere 8.5 lbs, and will take up around 15 megabytes of hard disk space. It is designed for, and, in my opinion, only needed by, commercial software developers. It will let you write programs that run under OS2 or use EMS memory for ordinary DOS. It is a superb development system, flawed at present only by its inability to work very well with Windows 3.0.

The environment that you usually use in the PDS is an extension of the QuickBASIC environment, called, naturally enough, the QuickBASIC extended or QBX. It is similar enough so that you won't have any problems shifting. Unlike with QB, however, the QBX environment takes advantage of extended memory. This lets you develop much larger programs than is possible under QB. The second environment is called the Microsoft Programmers workbench, which is used if you program in other languages or want to create programs that run under OS2. It uses the Microsoft Codeview debugger, and lets you mix programming languages at will.

The most important features of the PDS are:

ISAM—the indexed sequential access method. This allows you develop very sophisticated file handling programs, by keeping tables and indices that relate to a given file. Of course, if you master the method of binary trees given in chapter 7, this will seem less miraculous. ISAM sets up your files as giant tables—think of each row as corresponding to a record—and lets you access records by the contents of each field. A good way to think of an ISAM file is that it is an extremely well indexed random access file; although, in fact, it is stored sequentially.

Language enhancements: New commands like REDIM PRESERVE that let you change the size of an array without erasing the data. A currency data type, and functions for handling this new type. Functions that do the work of some of the DOS interrupts. (But now you know how to write your own!) You can find out what kind of device timeout occurred though the ERDEV$ and ERDEV functions. Improved error handling, because error traps can work at the procedure level, and not just in module level code.

Toolboxes: Contain functions that do everything from financial analysis to handle matrices. There are toolboxes for screen handling, presentation graphics, window generation, and mouse support.

Most importantly:

Performance enhancements: You can have more than a single 64K segment for strings. You can use overlays so your programs can be larger than fit in memory at a given time. Finally, the PDS gives programs that run faster and can be smaller; you can choose to leave out unused library features from the compiled code.

My impression from testing random programs is that you can expect around a 20% improvement in both speed and size for the same program developed in the PDS vs the QB environment. Of course, all this comes at a cost: $495, and you better have a 386 with a couple of megabytes of memory.

Where to go from here

I don't know if this will come as a surprise, but even a book this long couldn't cover all the features and powers of QuickBASIC. There are a couple of topics, like using QB as a standalone compiler (without using the QB environment) that the supplied manuals cover very well—if you're interested, you should look there for them. Actually, you might want to skim the manual at this point; it's written from a slightly different point of view, so you may find it occasionally useful.

The real point of this final section is to give you some other suggestions on where to go from here. I think the first thing you should do is learn more about DOS and batch programming. DOS's batch language will appear to you as nothing more than a very primitive form of BASIC, and so batch programming should take you only a couple of hours to master. You can find this information in the DOS manual. If you like the power of batch languages and would like to go further with them, Mansfield Software (1-203-429-8402) sells a full featured batch language called *REXX* that I like a lot.

The first place that I turn to for ideas for fun, and interesting programs, is A. K. Dewdney's "Computer Recreations" column in *Scientific American*. Many of the earlier columns were collected in *The Armchair Universe*, (Freeman, 1988). If you want to find out more about graphics, or need other ideas on interesting programs, then this is a good place to start.

If you want to use the methods described in chapter 8 and need to look up the formulas for various curves try *A Catalog of Special Plane Curves* by J. Lawrence, (Dover, 1978). For fractals go on to *The Science of Fractal Images* edited by H.O. Peitgen and D. Saupe, (Springer-Verlag, 1988). Be prepared for the mathematics, however. A book on IBM graphics which uses relatively little math is J. McGregor and A. Watt, *The Art of Graphics for the IBM PC*, (Addison-Wesley, 1986). Unfortunately, all the programs are written in interpreted BASIC, which doesn't allow recursion. Hence they're occasionally very hard to read. The text is clear though.

To understand the most modern ciphers, those devised, say, within the last 20 years, also requires a fair amount of mathematics. However, you can start with D. Kahn's massive *The Code-Breakers*, (Macmillan, 1967). This book is a lot of fun—both to read from cover to cover, or dip into occasionally. Next, try F. Higenbottom's *Codes and Ciphers*, (David MacKay, 1973). Finally, for those who don't mind a bit of math, try *Elementary Cryptoanalysis* by A. Sinkov, (Mathematical Association of America, 1968) or *Elementary Cryptoanalysis* by H. Gaines, (Dover, 1956).

For more on algorithms, a standard textbook is R. Sedgewick's *Algorithms*, (Addison-Wesley, 1988).

The real question, however, is whether you want to stick only to QuickBASIC—whether you want to move on to mixed language programming, or leave BASIC entirely. First, you have to decide whether you want to learn assembly language. Programming in assembler is tedious, and you have to buy an assembler, but it can give you incredible power. Rewrite a crucial part of a QB program as a CALL to an assembly language program and the speedups sometimes seem miraculous. My favorite book on assembler is R. Lafore's *Assembly Language Primer*, (New American Library, 1984). Be warned: at present, all assemblers are unpleasant—none have an integrated environment like QB.

There are three other computer languages that are popular for microcomputers. Pascal is the main instructional language of the universities, although it's far from being my favorite language. Since QuickBASIC borrowed many of the best features of Pascal, you won't find the transition very difficult, however. The best Pascal compiler is Borland International's *Turbo Pascal*. If you are a student, then call Borland to find out if their special student price is still available. There are literally hundreds of books on *Turbo Pascal* available, and to be quite honest, I don't know them well enough to have a favorite. I do have a favorite book on Pascal—in fact it's my favorite book on programming. I've learned a lot from it, and it's influenced how I explained structured programming here and in my classes. It's *Oh Pascal* by Doug Cooper and Michael Clancy, (Norton, 1985). Unfortunately, there's only a small add-in book for Turbo Pascal.

The language of professional programmers seems, more and more, to be C. This language was invented at Bell Labs to write operating systems, and so is much closer to assembly language than any other popular language. Because it's close to assembly language, it's fast. But for the very same reason, it's cryptic; it's been described as a "write only language." I don't particularly like C, but I do find myself using it more often than assembly language. I suggest that before you go out and buy a C compiler (a QB programmer will find Microsoft's Quick C, the easiest to get used to, and it comes with an excellent tutorial) that you look at, *The Art of C Programming* by Robin Jones and Ian Stewart, (Springer Verlag, 1987). It's very well written, and after you finish it you should be able to decide if you want to program in C at all.

My current favorite language (other than my dream version of QB—I expect, at the rate QB gets better, that version 5.5 will be it), is Modula-2. This was invented by Niklaus Worth, the inventor of Pascal, as a successor to Pascal. The best Modula-2 version that I know of, and one of the best implementations of any computer language for that matter, is *TopSpeed Modula-2* from Jensen and Partners (1-800-543-5202).

A

APPENDIX

Options on starting QuickBASIC

QB /RUN *program*	This loads QuickBASIC and runs the program named.
QB /B	Use when you have a monochrome monitor attached to a color graphics (CGA) card (combine with /NOHI for laptops).
QB /G	This is also only useful with a CGA. Allows a faster screen updating - at the risk of some snow.
QB / H	Gives the highest resolution possible.
QB / NOHI	Lets you use a monochrome monitor that doesn't support high intensity. (Mostly laptops - see /B.)
QB /AH	Allows dynamic arrays to be larger than 64K. (Don't try this without 640K, or if you use many memory-resident programs!)
QB /MBF	Use when converting a program to Microsoft (old fashioned) format numbers.

QB / L *NameOfLibrary*	Loads the quick library specified. If no library is given, loads QB.QLB (which contains the DOS interrupts c.f. chapter 11).
QB /CMD *a string*	Passes the string to the COMMAND$ function. This must be the last option.
QB /C:*size of buffer*	Sets the size of the buffer for communicating via a serial port.

Most of these options, that are not mutually exclusive, can be combined. Example: QB /L /H /C:2048 /CMD "Test" would load the QB.QLB library, display the highest resolution possible, set the com buffer to 2048, and pass the string "Test" to COMMAND$; or QB /B /NOHI is the best way to run QB with a laptop LCD.

B
APPENDIX

Interpreted BASIC versus QuickBASIC

The first thing you must do in converting an interpreted BASIC (BASICA or GW-BASIC) program to QuickBASIC is to make sure that it's been saved in ASCII form, using the SAVE "fname," A option. (The normal way an interpreted BASIC program saves files is incompatible with QuickBASIC, much like choosing the fast load option, in a save file command, is incompatible with other word processors.)

The following statements in interpreted BASIC are editing commands or commands concerned with running a program. Their counterparts in QuickBASIC are done from one of the menus. (In particular the editing facilities of QuickBASIC are far superior.)

AUTO (Line numbers aren't needed in QuickBASIC anyway.)
CONT (Done from the RUN menu, or via the F5 shortcut. Usually done after the **R**estart command or a break—Ctrl+Break, or a debugging break.)
DELETE (Select the block and then hit the DEL key or use Cu**T** from the Edit menu to move it to the clipboard.)

EDIT (Move the cursor to the line, and then edit to your heart's content.)
LIST (Make the View window active and scroll.)
LLIST (Use PRINT on the File menu to print the active window.)
LOAD (Done from the File menu.)
MERGE (Done from the full File menu.)
NEW (Done from the File menu.)
RENUM (Don't need line numbers in the first place.)
RUN (SHIFT + F5 or Start on the Run menu.)
SAVE (Again, done from the File menu.)

The following command has no counterpart in QuickBASIC.

MOTOR (I've never seen a PC with a cassette tape. Have you?)

The following commands have been superceded:

DEF USR and **USR**	These commands set up a machine language routine in interpreted BASIC. QuickBASIC's facilities are far superior. (See chapter 11.)

The following statements work a bit differently:

BLOAD, BSAVE The statements theoretically work the same—what they may do may be different. QuickBASIC and interpreted BASIC use different memory locations for similar functions. (See PEEK and POKE.)

CALL Calls a subprogram, not an assembly language routine.

CHAIN QuickBASIC does not support the ALL, DELETE, MERGE, or using a line number.

CLEAR The first position is used in interpreted BASIC for the DATA segment. This isn't used in QuickBASIC. (That's why you need a comma to set the stack size.)

COMMON When used to pass values of variables via CHAIN, must appear before any executable statement.

DEFiner (DEFINT, DEFSNG etc . . .) In QuickBASIC these must appear on a physical line before any variables that you want to affect. (In interpreted BASIC the order can be affected by branches.)

DEF FN Order as for a DEFiner

DIM Order as for a DEFiner or DEF FN

PEEK and **POKE** Be very careful with any program that PEEKs or POKEs. The locations used by QuickBASIC may be different.

RESUME If an error occurs in using a DEF FN function, QuickBASIC continues execution at the line that defines the function.

SYSTEM In QuickBASIC this ends a program and takes you back to DOS or the QuickBASIC environment, depending on whether the program is stand alone or not.

RUN The , **R** option that keeps files open isn't supported.

C

APPENDIX

Differences between QB 4.5 and earlier versions

QuickBASIC 4.5 is an evolutionary upgrade to QuickBASIC 4.0. QuickBASIC 4.0, on the other hand, is a revolutionary change from earlier versions of Quick-BASIC. The only new commands in QuickBASIC 4.5 are the extremely useful SLEEP command and the less useful UEVENT, ON UEVENT pair that is used for specialized event trapping. (These require some knowledge of assembly language.)

The main difference between 4.5 and 4.0 is the help system. In QuickBASIC 4.5, almost anything you'll want to know about QuickBASIC is on line—available with a few keystrokes. In QuickBASIC 4.0, most of this information was available in one of the manuals.

The other changes are in the menus and the $INCLUDE command. The most important is the $INCLUDE command. In earlier versions of QB you could $INCLUDE files containing executable commands. In QuickBASIC 4.5, $INCLUDE is only for non-executable statements (DECLARE and type defining statements mostly). The other change is the distinction between "easy" menus and "full" menus.

QB 4.5 is a bit bigger then QB 4.0. So if the memory in your machine was barely sufficient to handle QB 4.0, then you'll likely not be able to handle QB 4.5. (If you were using a TSR program, like Sidekick, then you may have to eliminate it.)

For more details on the changes, please see Appendix B of the *Programming in BASIC* manual.

D

APPENDIX

Shortcut keys and editing commands

Most of this information is available on-line. Go to Contents in the **Help** menu.

F1 Used when cursor is on a specific keyword. Gives help on that command.
Shift + F1 Ranges from "Help on Help" to help on a menu item. Gateway to the various help files through *hyperlinks*.
Alt + F1 Lets you move backwards to previous help screens.
F2 Opens a window listing modules with their associated SUB's and FUNCTION procedures. Allows you to edit a specific procedure or function.
Shift + F2 Moves to the main module or a function or procedure (alphabetically). Places it in the active view window.
Ctrl + F2 Moves backwards—opposite of Shift + F2.
F3 Repeats last find (Edit menu).
F4 Switches back and forth from the output screen to the development environment.
F5 Continue command. Used after a break or restart (Run menu).
Shift + F5 Starts program execution (Start on Run menu).
(Ctrl + F5) An undocumented version of Ctrl + F10

F6 Makes next window active.
Shift + F6 Makes previous window active.
F7 Executes program up to current cursor position.
F8 "Single steps" through program.
F9 Toggles breakpoints on and off (Debug menu).
Shift + F9 Brings up the "instant watch" dialog box (Debug menu).
F10 Also single steps, but treats procedures and functions as single objects ("steps through").
Shift + F10 Used in connection with the History On option in the full Debug menu. Steps forward through the last 20 statements recorded.
Ctrl + F10 Switch active window between original size and full screen.
Alt + Enlarges active window one line.
Alt − Shrinks active window one line.

Editing keys

First off, you can undo any changes made to a line, provided the cursor hasn't left the line. This is done with Alt + Backspace, or the Undo option on the (full) Edit menu.

The following table describes the editing keys, the last column giving the WordStar equivalent—which QuickBASIC also supports.

Character left	(left arrow)	Ctrl + S
Character right	(right arrow)	Ctrl + D
Line up	(up arrow)	Ctrl + E
Line down	(down arrow)	Ctrl + X
First indentation level	Home	
Beginning of line		Ctrl + Q+S
End of line	End	Ctrl + Q+D
Beginning of next line	Ctrl + Enter	Ctrl + J
Page Up	Page Up	Ctrl + R
Page down	Page Down	Ctrl + C
Top of window		Ctrl + Q+E
Bottom of window		Ctrl + Q+X
Top of document	Ctrl + Home	Ctrl + Q+R
End of document	Ctrl + End	Ctrl + Q+C
Word left	Ctrl +	Ctrl + A
Word right	Ctrl +	Ctrl + F
Screen left	Ctrl + Pg Up	
Screen right	Ctrl + Pg Down	

(Notice the WordStar commands form a diamond on the left hand side of the keyboard.)

Insert mode is toggled on and off by the insert key or Ctrl + V (the cursor changes from a _ to a ■). You can:

Insert a line below current line	End + Enter
Insert a line above current line	Home + Ctrl + N

The tab key inserts spaces (number determined via Display on the Options menu).

Simple deletions are done via:

Character at cursor	Del or Ctrl+G
Character before cursor	Backspace or Ctrl+H
Word	Ctrl+T

You can also:

Delete the whole line	Ctrl+Y

or

Delete from the cursor to the end of the line: Ctrl+Q+Y

Both of these actually place the deleted text in the clipboard. The clipboard is used to "cut, copy and paste." Once text has been placed here you can "paste" it by moving the cursor and hitting Shift+Ins or using Paste on the Edit menu.

All other ways to place text in the clipboard requires selecting it. This is done by holding down the Shift key while using any of the cursor movement keys given above. As you move the cursor in the way indicated above you select more (or less) text. Selected text is either highlighted or appears in reverse video, depending on your monitor. Once text is selected you can:

Completely delete it	(press the Del key)
Indent it	(Tab)
Cut it to the clipboard	(Shift+Del or use the Edit menu)
Copy it to the clipboard	(Ctrl+Insert or use the edit menu)

Because text remains in the clipboard until replaced by another operation, you can even paste multiple copies. To deselect text, hit any arrow key. Doing this is essential. This is because while text is selected any key you press will replace the selected text. (As you can imagine this is extremely frustrating—lean on the keyboard and all the selected text is gone!)

You can place up to four markers within your document or with the help files. This allows you to move rapidly within the document, or back to a specific piece of "help" information. To do this move the cursor to the place you want to mark and type:

Ctrl+K+n (where n is a number from 0 to 3)

to move to the marker enter:

Ctrl+Q+n

Finally, you can scroll the text in the active window via:

Up one line	Ctrl +	Ctrl + W
Down a line	Ctrl +	Ctrl + Z
Page up	Page Up	Ctrl + R
Page down	Page Down	Ctrl + C
Window left	Ctrl + Page Up	
Window right	Ctrl + Page Down	

E
APPENDIX

Mice

As I mentioned earlier I find the keyboard shortcuts easier.

First some terminology: the "mouse cursor" is a block (■) that moves as you move the mouse. "To click" means to press a mouse button once. "To double click" means press the same button twice in quick succession. "To drag" means to hold down the left mouse button and move the mouse. Using this terminology, here are the mouse equivalents for many of the commands:

To choose a menu item	Move mouse cursor to the menu and click left button. Move mouse cursor to item and click again.
To close a menu	Move mouse cursor off menu and click left button.
Make a window active	Move mouse cursor inside window and click left button.
Expand or shrink window	Move mouse cursor to title bar and drag.
Expand window to full size	Double click left button anywhere on title bar. (Also there is a maximize box at the right side of the title bar—can click left button once here.)

Scroll text	For a single line or character click left button with mouse cursor at the appropriate scroll "arrow." Otherwise, drag the scroll box to roughly the position on scroll bar. To scroll one page, move mouse cursor to scroll bar and click the left button.
To load a program or file	Move mouse cursor to the file name (as revealed in a dialog box) and double click on it. (Also can click once and hit enter.)

The right mouse button has two possibilities (change via the option menu). The default (usual) gives context sensitive help. The other is to change it to execute a program up to the line the mouse cursor is on. Finally, Shift+ a right mouse click gives instant watch (= Shift+F9).

F

APPENDIX

Reserved words, ASCII codes and scan codes

ABS
ACCESS
ALIAS
AND
ANY
APPEND
AS
ASC
ATN
BASE
BEEP
BINARY
BLOAD
BSAVE
BYVAL
CALL
CALLS
CASE
CDBL
CDECL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLNG
CLOSE
CLS
COLOR
COM
COMMAND$
COMMON
CONST
COS
CSNG
CSRLIN
CVD
CVDMBF
CVI
CVL
CVS
CVSMBF
DATA
DATE$
DECLARE
DEF
DEFDBL
DEFINT
DEFLNG
DEFSNG
DEFSTR
DIM
DO
DOUBLE
DRAW

ELSE
ELSEIF
END
ENDIF
ENVIRON
ENVIRON$
EOF
EQV
ERASE
ERDEV
ERDEV$
ERL
ERR
ERROR
EXIT
EXP
FIELD
FILEATTR
FILES
FIX
FOR
FRE
FREEFILE
FUNCTION
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT$
INSTR
INT
INTEGER
IOCTL
IOCTL$
IS
KEY
KILL
LBOUND
LCASE$
LEFT$
LEN
LET
LINE
LIST
LOC
LOCAL
LOCATE
LOCK
LOF
LOG
LONG
LOOP
LPOS
LPRINT
LSET
LTRIM$
MID$
MKD$
MKDIR
MKDMBF$
MKI$
MKL$
MKS$
MKSMBF$
MOD
NAME
NEXT
NOT
OCT$
OFF
ON
OPEN
OPTION
OR
OUT
OUTPUT
PAINT
PALLATTE
PCOPY
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PSET
PUT
RANDOM
RANDOMIZE
READ
REDIM
REM
RSET
RESTORE
RESUME
RETURN
RIGHT$
RMDIR
RND
RSET
RTRIM$
RUN
SADD
SCREEN
SEEK
SEG
SELECT
SETMEM
SGN
SHARED
SHELL
SIGNAL
SIN
SINGLE
SLEEP
SOUND
SPACE$
SPC
SQRP
STATIC
STEP
STICK
STOP
STR$
STRIG
STRING
STRING$
STRING$
SUB
SYSTEM
TAB
TAN
THEN
TIME$
TIMER
TO
TROFF
TRON
TYPE
UBOUND
UCASE$
UEVENT
UNLOCK
UNTIL
USING
VARPTR
VARPTR$
VIEW
WAIT
WEND
WHILE
WIDTH
WINDOW
WRITE
XOR

Remember that no variable can begin with FN.

The ASCII Codes and the SCAN codes are available on line—use the contents page on the help system. The SCAN codes are used when trapping a key via the KEY statement. The first column given in the help system is the key pressed, and the second is the scan code. So, for example, to trap a 2 use scan code 3!

G
APPENDIX

Error messages

This is a list of those "run-time" error messages that you may need to trap. For a complete list, including compile time errors, see Appendix I of the *Programming in BASIC* manual.

11	division by 0	
24	Device timeout	(Printer not on line?)
25	Device fault	(general hardware problem)
27	Out of paper	
52	Bad file name	(Form of file name not allowed)
53	File not found	(Check the full path name)
54	Bad file mode	
55	File already open	(Close it!)
57	Device I/O error	(Problems with the disk?)
58	File already exists	(You can't reNAME it the way you want.)
61	Disk full	(If you're lucky you can SHELL to FORMAT.COM.)
67	Too many files	A directory or sub-directory has limits on the number of files allowed (112 and 255 respectively).

68	Device unavailable	(Turn it on.)
69	COM buffer overflow	(Empty it more often or use the QB /C option when starting QuickBASIC to enlarge it.)
70	Permission denied	(A write-protect tab may be on.)
71	Disk not ready	(Put the disk in and make sure the drive door is closed.)
72	Disk media error	(Corrupted disk—replace.)
75	Path/File access error	
and		(Check the full path name.)
76	Path not found	

H

APPENDIX

Customizing QuickBASIC

This is done from the Display item on the Option menu. You can set tabs, change colors or remove the scroll bars (useful if you don't have a mouse). For example, I find it useful to change the tab stops setting to 5. The changes you make are saved in the QB.INI file.

Most importantly you can use the Path option to tell QB where to find library files and the help files. This lets you make changes without having to run the setup program again.

I

APPENDIX

The GOTO and other artifacts

QuickBASIC (for compatibility) still allows certain commands that are usually best forgotten. In most cases they have long since been superceded. The most egregious offender is the naked GOTO. This command sends processing to a statement indicated by the label or line number. There is almost never any need to do this. Why would you want to jump unconditionally? You should always have a reason. And, since QuickBASIC has the various forms of the EXIT command (EXIT DO, EXIT FOR) programs like:

```
FOR I = 1 TO 100
    PRINT I
    IF INKEY$ < > " " THEN GOTO Ych
NEXT I
Ych:
    PRINT "Abnormal exit!:
```

can easily be rewritten using the EXIT FOR. (The only time that a GOTO is ever needed is when you're deep in a nested loop and want to get out because of an abnormal condition.)

Similarly, early versions of BASIC (AppleSoft, for example) had neither a DO LOOP nor a WHILE loop. You were forced to mimic these useful statements by the following kind of ugly and confusing code:

```
10 IF X > 5 THEN GOTO 100
20  .
30  .
40  .
.
.
90 GOTO 10
100 END
```

Isn't it cleaner to say?

```
DO WHILE X < = 5
    .
    .
    .
LOOP
```

which works the same way. Except for error trapping you never need a GOTO in QuickBASIC.

Index

Though C may be gaining popularity among software developers in all areas of computing, BASIC is still the predominant general-purpose programming language—led by Microsoft's QuickBASIC with an installed user base of more than half a million. Now, with ***QuickBASIC 4.5,*** you can quickly master the sophisticated new look and feel of QuickBASIC Version 4.5.

In addition to providing a complete introduction to 4.5 data types, procedures and subroutines, this comprehensive guide includes coverage of:

- QuickBASIC's new indexing, sorting, and file security capabilities (encryption)
- Logic structures such as CASE and IF-THEN-ELSE, and data structures such as arrays and records
- Working with ciphers, interacting with DOS, using ISAM support, optimizing binary-trees and fractals

. . . and much more. Cornell even explains recursion—a powerful programming technique never before available in BASIC. Numerous appendices include topics such as comparing other forms of BASIC to QuickBASIC, comparisons to earlier versions, reserved words, error messages, and more.

IBM Programming		
Beginner	[✓ Intermediate]	Advanced

A hands-on tutorial and desktop reference for the latest version of Microsoft's QuickBASIC

Featuring hundreds of plug-in code examples and listings for DOS-based PCs

About the Author

The author of three computer books, Gary Cornell, Ph.D., is an associate professor of mathematics at the University of Connecticut and formerly a Visiting Scientist at IBM's Thomas J. Watson Research Center.

0791 U.S. $22.95

ISBN 0-8306-3782-6

90000

9 780830 637829

Windcrest®/McGraw-Hill
Blue Ridge Summit, PA 17294-0850